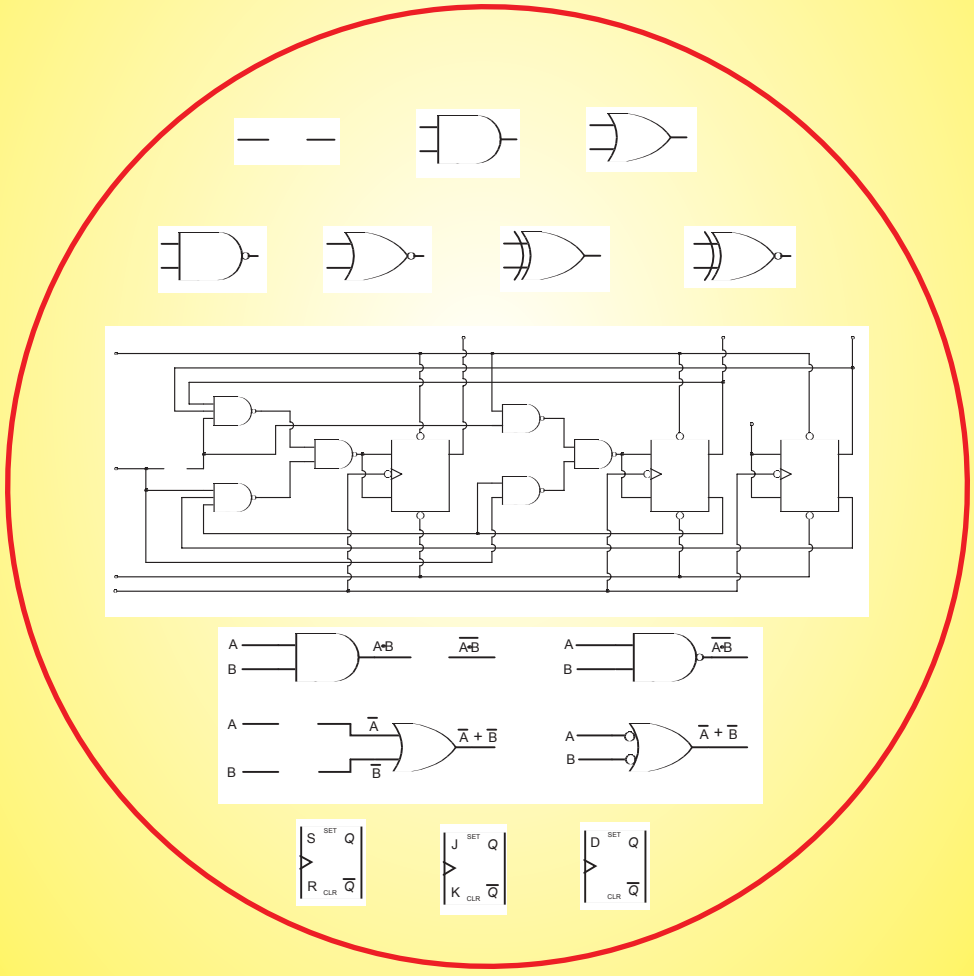
Digital Circuit Analysis and Design

with an Introduction to CPLDs and FPGAs

Steven T. Karris
Editor



Orchard Publications
www.orchardpublications.com

VISIT…

Digital Circuit Analysis and Design

with an Introduction to CPLDs & FPGAs

Students and working professionals will find *Digital Circuit Analysis and Design with an Introduction to CPLDs and FPGAs*, to be a concise and easy-to-learn text. It provides complete, clear, and detailed explanations of the state-of-the-art electronic digital circuits. All topics are illustrated with many real-world examples.

This text includes the following chapters and appendices:

• Common Number Systems and Conversions • Operations in Binary, Octal, and Hexadecimal Systems • Sign Magnitude and Floating Point Arithmetic • Binary Codes • Fundamentals of Boolean Algebra • Minterms and Maxterms • Combinational Logic Circuits • Sequential Logic Circuits • Memory Devices • Advanced Arithmetic and Logic Operations • Introduction to Field Programmable Devices • Introduction to the ABEL Hardware Description Language • Introduction to VHDL • Introduction to Verilog • Introduction to Boundary-Scan Architecture

Each chapter contains numerous practical applications. This is a design-oriented text.

Steven T. Karris is the president and founder of Orchard Publications. He earned a bachelors degree in electrical engineering at Christian Brothers University, Memphis, Tennessee, a masters degree in electrical engineering at Florida Institute of Technology, Melbourne, Florida, and has done post-master work at the latter. He is a registered professional engineer in California and Florida. He has over 35 years of professional engineering experience in industry. In addition, he has over 30 years of teaching experience that he acquired at several educational institutions as an adjunct professor. He was formerly with UC Berkeley Extension.

Orchard Publications

Visit us on the Internet
www.orchardpublications.com
or email us: info@orchardpublications.com

ISBN 0-9744239-5-5

$49.95 U.S.A.

Digital Circuit Design with an Introduction to CPLDs and FPGAs

Steven T. Karris
Editor

Orchard Publications
www.orchardpublications.com

Digital Circuit Design with an Introduction to CPLDs and FPGAs

Copyright " 2005 Orchard Publications. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher.

Direct all inquiries to Orchard Publications, info@orchardpublications.com

Product and corporate names are trademarks or registered trademarks of Xilinx, Inc., Altera, Inc. Cypress Semiconductor, Lattice, Inc., and Atmel, Inc. They are used only for identification and explanation, without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Library of Congress Control Number (LCCN) 2005929326

Copyright TX 5-612-942

ISBN 0-9744239-5-5

Disclaimer

The author has made every effort to make this text as complete and accurate as possible, but no warranty is implied. The author and publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this text.

Preface

This book is an undergraduate level textbook presenting a thorough discussion of state-of-the-art digital devices and circuits. It supplements our *Electronic Devices and Amplifier Circuits*, ISBN 0-9744239-4-7. It is self-contained; begins with the basics and ends with the latest developments of the digital technology. The intent is to prepare the reader for advanced digital circuit design and programming the powerful Complex Programmable Logic Devices (CPLDs), and Field Programmable Gate Arrays (FPGAs).

The prerequisites for this text are just basic high-school math; Accordingly, it can be read and understood by high-school seniors, trade-school, community college, and 4-year university students. It is ideal for self-study.

The author and contributors make no claim to originality of content or of treatment, but have taken care to present definitions, statements of physical laws, theorems, and problems.

Chapter 1 is an introduction to the decimal, binary, octal, and hexadecimal numbers, their representation, and conversion from one base to another. Chapter 2 presents an introduction to arithmetic operations in binary, octal, and hexadecimal numbers. The tens complement and nines complements in the decimal system and the twos complement and ones complements in the binary system are discussed and illustrated with numerous examples. Chapter 3 begins with an introduction to sign magnitude representation of binary numbers. It concludes with a discussion on floating point arithmetic for representing large numbers and the IEEE standard that specifies single precision (32 bit) and double precision (64 bit) floating point representation of numbers.

Chapter 4 describes the most commonly used binary codes. The Binary Coded Decimal (BCD), the Excess-3 Code, the 2*421 Code, the Gray Code, and the American Standard Code for Information Interchange (ASCII) code are introduced as well as the use of parity bits. Chapter 5 begins with the basic logic operations and continues with the fundamentals of Boolean algebra and the basic postulates and theorems as applied to electronic logic circuits. Truth tables are defined and examples are given to illustrate how they can be used to prove Boolean algebra theorems or equivalent logical expressions. Chapter 6 introduces the standard forms of expressing Boolean functions; the minterms and maxterms, also known as standard products and standard sums respectively. A procedure is also presented to show how one can convert one form to the other. This topic is essential in understanding the programming of Programmable Logic Arrays (PLAs) discussed in Chapter 11.

Chapter 7 is an introduction to combinational logic circuits. It begins with methods of implementing logic diagrams from Boolean expressions, the derivation of Boolean expressions from logic diagrams, input and output waveforms, and the use of Karnaugh maps for simplifying Boolean expressions. Chapter 8 is an introduction to sequential logic circuits. It begins with a

discussion of the different types of flip flops, and continues with the analysis and design of binary counters, registers, ring counters, and ring oscillators. Chapter is an introduction to computer memory devices. We discuss the random-access memory (RAM), read-only memory (ROM), row and column decoders, memory chip organization, static RAMs (SRAMs) dynamic RAMs (DRAMs), volatile, nonvolatile, programmable ROMs (PROMs), Erasable PROMs (EPROMs), Electrically Erasable PROMs (EEPROMs), flash memories, and cache memory. Chapter 10 begins with an introduction to the basic components of a digital computer. It continues with a discussion of the basic microprocessor operations, and concludes with the description of more advanced arithmetic and logic operations.

We consider Chapter 11 as the highlight of this text. It is an introduction to Field Programmable Devices (FPDs), also referred to as Programmable Logic Devices (PLDs). It begins with the description and applications of Programmable Logic Arrays (PLAs), continues with the description of Simple PLDs (SPLDs) and Complex PLDs (CPLDs), and concludes with the description of Field Programmable Gate Arrays (FPGAs).

This text includes also four appendices; Appendix A is an overview of the Advanced Boolean Equation Language (ABEL) which is an industry-standard Hardware Description Language (HDL) used in Programmable Logic Devices (PLDs). Appendix B describes the VHSIC Hardware Description Language briefly referred to as VHDL. This language was developed to be used for documentation, verification, and synthesis of large digital designs. Appendix C introduces the Verilog Hardware Description Language (HDL). Like VHDL introduced in Appendix B, Verilog is a programming language used to describe a digital system and its components. Appendix D is a brief discussion on the boundary-scan architecture and the new technology trends that make using boundary-scan essential for the reduction in development and production costs.

This is our eighth science and electrical and computer engineering-related text. My associates, contributors, and I have a mission to produce substance and yet inexpensive texts for the average reader. Our texts are very popular with students and working professionals seeking to enhance their knowledge and prepare for the professional engineering examination. We are working with limited resources and our small profits realized after large discounts to the bookstores and distributors, are reinvested in the production of more texts. To maintain our retail prices as low as possible, we avoid expensive and fancy hardcovers.

Like any other new text, the readers will probably find some mistakes and typo errors for which we assume responsibility. We will be grateful to readers who direct these to our attention at info@orchardpublications.com. Thank you.

Orchard Publications
Fremont, California 94538-4741
United States of America
www.orchardpublications.com
info@orchardpublications.com

Table of Contents

Chapter 1

Common Number Systems and Conversions

Decimal, Binary, Octal, and Hexadecimal Systems....1-1
Binary, Octal, and Hexadecimal to Decimal Conversions....1-3
Decimal to Binary, Octal, and Hexadecimal Conversions....1-3
Binary-Octal-Hexadecimal Conversions....1-7
Summary....1-9
Exercises....1-11
Solutions to End-of-Chapter Exercises....1-12

Chapter 2

Operations in Binary, Octal, and Hexadecimal Systems

Binary System Operations....2-1
Octal System Operations....2-2
Hexadecimal System Operations....2-5
Complements of Numbers....2-6
Tens-Complement....2-7
Nines-Complement....2-7
Twos-Complement....2-8
Ones-Complement....2-9
Subtraction with Tens- and Twos-Complements....2-10
Subtraction with Nines- and Ones-Complements....2-11
Summary....2-14
Exercises....2-16
Solutions to End-of-Chapter Exercises....2-18

Chapter 3

Sign Magnitude and Floating Point Arithmetic

Signed Magnitude of Binary Numbers....3-1
Floating Point Arithmetic....3-2
The IEEE Single Precision Floating Point Arithmetic....3-3
The IEEE Double Precision Floating Point Arithmetic....3-7
Summary....3-9
Exercises....3-10
Solutions to-End-of-Chapter Exercises....3-11

Chapter 4

Binary Codes

Encoding 4-1
Binary Coded Decimal (BCD) 4-1
The Excess-3 Code 4-2
The 2*421 Code 4-3
The Gray Code 4-4
The American Standard Code for Information Interchange (ASCII) Code 4-5
The Extended Binary Coded Decimal Interchange Code (EBCDIC) 4-8
Parity Bits 4-8
Error Detecting and Correcting Codes 4-9
Cyclic Codes 4-9
Summary 4-14
Exercises 4-16
Solutions to End-of-Chapter Exercises 4-17

Chapter 5

Fundamentals of Boolean Algebra

Basic Logic Operations 5-1
Fundamentals of Boolean Algebra 5-1
Postulates 5-1
Theorems 5-2
Truth Tables 5-3
Summary 5-5
Exercises 5-7
Solutions to End-of Chapter Exercises 5-8

Chapter 6

Minterms and Maxterms

Minterms 6-1
Maxterms 6-2
Conversion from One Standard Form to Another 6-3
Properties of Minterms and Maxterms 6-4
Summary 6-9
Exercises 6-10
Solutions to End-of-Chapter Exercises 6-12

Chapter 7

Combinational Logic Circuits

Implementation of Logic Diagrams from Boolean Expressions 7-1
Obtaining Boolean Expressions from Logic Diagrams 7-9
Input and Output Waveforms 7-11
Karnaugh Maps (K-maps) 7-12
K-map of Two Variables 7-12
K-map of Three Variables 7-14
K-map of Four Variables 7-14
General Procedures for Using a K-map of n Squares 7-16
Don't Care Conditions 7-20
Design of Common Logic Circuits 7-21
Parity Generators/Checkers 7-21
Digital Encoders 7-23
Decimal-to-BCD Encoder 7-26
Digital Decoders 7-28
Equality Comparators 7-32
Multiplexers and Demultiplexers 7-36
Arithmetic Adder and Subtractor Logic Circuits 7-42
Summary 7-48
Exercises 7-50
Solutions to End-of-Chapter Exercises 7-53

Chapter 8

Sequential Logic Circuits

Introduction to Sequential Circuits 8-1
Set-Reset (SR) Flip Flop 8-1
Data (D) Flip Flop 8-4
JK Flip Flop 8-5
Toggle (T) Flip Flop 8-6
Flip Flop Triggering 8-7
Edge-Triggered Flip Flops 8-8
Master / Slave Flip Flops 8-8
Conversion from One Type of Flip Flop to Another 8-11
Analysis of Synchronous Sequential Circuits 8-13
Design of Synchronous Counters 8-22
Registers 8-27
Ring Counters 8-32
Ring Oscillators 8-35

Summary 8-36
Exercises 8-39
Solutions to End-of-Chapter Exercises 8-42

Chapter 9

Memory Devices

Random-Access Memory (RAM) 9-1
Read-Only Memory (ROM) 9-3
Programmable Read-Only Memory (PROM) 9-6
Erasable Programmable Read-Only Memory (EPROM) 9-7
Electrically-Erasable Programmable Read-Only Memory (EEPROM) 9-8
Flash Memory 9-8
Cache Memory 9-9
Virtual Memory 9-9
Scratch Pad Memory 9-10
Summary 9-11
Exercises 9-13
Solutions to End-of-Chapter Exercises 9-14

Chapter 10

Advanced Arithmetic and Logic Operations

Computers Defined 10-1
Basic Digital Computer System Organization and Operation 10-2
Parallel Adder 10-4
Serial Adder 10-5
Overflow Conditions 10-6
High-Speed Addition and Subtraction 10-9
Binary Multiplication 10-10
Binary Division 10-13
Logic Operations of the ALU 10-14
Other ALU functions 10-15
Summary 10-16
Exercises 10-18
Solutions to End-of-Chapter Exercises 10-19

Chapter 11

Introduction to Field Programmable Devices

Programmable Logic Arrays (PLAs) 11-1
Programmable Array Logic (PAL) 11-5
Complex Programmable Logic Devices (CPLDs) 11-6
The Altera MAX 7000 Family of CPLDs 11-7
The AMD Mach Family of CPLDs 11-12
The Lattice Family of CPLDs 11-14
Cypress Flash370 Family of CPLDs 11-15
Xilinx XC9500 Family of CPLDs 11-20
CPLD Applications 11-30
Field Programmable Gate Arrays (FPGAs) 11-36
SRAM-Based FPGA Architecture 11-37
Xilinx FPGAs 11-37
Atmel FPGAs 11-40
Altera FPGAs 11-40
Lattice FPGAs 11-42
Antifuse-Based FPGAs 11-43
Actel FPGAs 11-43
QuickLogic FPGAs 11-49
FPGA Block Configuration - Xilinx FPGA Resources 11-50
The CPLD versus FPGA Trade-Off 11-58
What is Next 11-58
Summary 11-61
Exercises 11-63
Solutions to End-of-Chapter Exercises 11-65

Appendix A

Introduction to ABEL Hardware Description Language

Introduction A-1
Basic Structure of an ABEL Source File A-1
Declarations in ABEL A-3
Numbers in ABEL A-5
Directives in ABEL A-6
The @alternate Directive in ABEL A-6
The @radix Directive in ABEL A-7
The @standard Directive in ABEL A-7
Sets in ABEL A-7
Indexing or Accessing a Set in ABEL A-8

Set Operations in ABEL A-9
Operators in ABEL A-11
Logical Operators in ABEL A-11
Arithmetic Operators in ABEL A-12
Relational Operators in ABEL A-12
Assignment Operators in ABEL A-13
Operator Priorities in ABEL A-13
Logic Description in ABEL A-14
Equations in ABEL A-14
Truth Tables in ABEL A-15
State Diagram in ABEL A-18
Dot Extensions in ABEL A-21
Test Vectors in ABEL A-22
Property Statements in ABEL A-23
Active-Low Declarations in ABEL A-23

Appendix B

Introduction to VHDL

Introduction B-1
The VHDL Design Approach B-1
VHDL as a Programming Language B-3
Elements B-4
Comments B-4
Identifiers B-4
Literal Numbers B-4
Literal Characters B-5
Literal Strings B-5
Bit Strings B-5
Data Types B-6
Integer Types B-6
Physical Types B-7
Floating Point Types B-8
Enumeration Types B-9
Arrays B-9
Records B-11
Subtypes B-11
Object Declarations B-12
Attributes B-13
Expressions and Operators B-14
Sequential Statements B-15

Variable Assignments B-15
If Statement B-16
Case Statement B-16
Loop Statements B-17
Null Statement B-19
Assertions B-19
Subprograms and Packages B-20
Procedures and Functions B-20
Overloading B-23
Package and Package Body Declarations B-24
Package Use and Name Visibility B-26
Structural Description B-26
Entity Declarations B-26
Architecture Declarations B-29
Signal Declarations B-30
Blocks B-30
Component Declarations B-32
Component Instantiation B-33
Behavioral Description B-33
Signal Assignment B-33
Process and the Wait Statement B-35
Concurrent Signal Assignment Statements B-38
Conditional Signal Assignment B-38
Selected Signal Assignment B-40
Organization B-41
Design Units and Libraries B-42
Configurations B-43
Detailed Design Example B-47

Appendix C

Introduction to Verilog

Description C-1
Verilog Applications C-2
The Verilog Programming Language C-2
Lexical Conventions C-6
Program Structure C-7
Data Types Defined C-9
Physical Data Types C-9
Abstract Data Types C-11
Operators Defined C-11

Binary Arithmetic Operators C-11
Unary Arithmetic Operators C-12
Relational Operators C-12
Logical Operators C-12
Bitwise Operators C-12
Unary Reduction Operators C-13
Other Operators C-13
Operator Precedence C-14
Control Statements C-15
Selection Statements C-15
Repetition Statements C-16
Other Statements C-16
Parameter Statements C-17
Continuous Assignment Statements C-17
Blocking Assignment Statements C-17
Non-Blocking Assignment Statements C-18
System Tasks C-19
Functions C-21
Timing Control C-22
Delay Control C-22
Event Control C-22
Wait Control C-22
Fork and Join Control C-23

Appendix D

Introduction to Boundary Scan Architecture

The IEEE Standard 1149.1 ..... D-1
Introduction D-1
Boundary Scan Applications D-3
Board with Boundary-Scan Components D-4
Field Service Boundary-Scan Applications D-5

Chapter 1

Common Number Systems and Conversions

This chapter is an introduction to the decimal, binary, octal, and hexadecimal numbers, their representation, and conversion from one base to another. The conversion procedures are illustrated with several examples.

1.1 Decimal, Binary, Octal, and Hexadecimal Systems

The familiar decimal number system has *base* or *radix* 10. It referred to as base 10 because it uses ten digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. These digits are referred to as the *coefficients* of the decimal system. Thus, in the decimal system the coefficients are multiplied by the appropriate powers of 10 to form a number. For example, the decimal number 58, 392.46 is interpreted as:

$$\begin{aligned} 58,392.46 &= 50,000 + 8,000 + 300 + 90 + 2 + 0.4 + 0.06 \\ &= 5\times 10^{4} + 8\times 10^{3} + 3\times 10^{2} + 9\times 10^{1} + 2\times 10^{0} + 4\times 10^{-1} + 6\times 10^{-2} \end{aligned}$$

In general, any number may be represented by a series of coefficients as:

$$A_nA_{n-1}A_{n-2}\ldots\ldots A_2A_1A_0.A_{-1}A_{-2}\ldots\ldots A_{-n}$$

In the decimal system, the A_k coefficients are the ten coefficients (zero through nine), and the subscript value k denotes the power of ten by which the coefficient must be multiplied. Thus, the last expression above can also be written as

$$A_n\cdot 10^{n} + A_{n-1}\cdot 10^{n-1} + A_{n-2}\cdot 10^{n-2} + \ldots + A_2\cdot 10^{2} + A_1\cdot 10^{1} + A_0\cdot 10^{0} + A_{-1}\cdot 10^{-1} + \ldots + A_{-n}\cdot 10^{-n}$$

Digital computers use the binary (base 2) system which has only two coefficients, 0 and 1. In the binary system each coefficient A_k is multiplied by 2^k. In general, a number of base or radix r with coefficients A_k is expressed as

$$A_n\cdot r^{n} + A_{n-1}\cdot r^{n-1} + A_{n-2}\cdot r^{n-2} + \ldots + A_2\cdot r^{2} + A_1\cdot r^{1} + A_0\cdot r^{0} + A_{-1}\cdot r^{-1} + \ldots + A_{-n}\cdot r^{-n} \qquad (1.1)$$

The number 110010.01 could be interpreted as a binary, or decimal or any other base number since the coefficients 0 and 1 are valid in any number with base 2 or above. Therefore, it is a recommended practice to enclose the number in parenthesis and write a subscript representing the base of the number. Thus, if the number 110010.01 is binary, it is denoted as

$$(110010.01)_2$$

But if it is a decimal number, it should be denoted as

$$(110010.01)_{10}$$

Two other numbers of interest are the *octal* (base 8) and *hexadecimal* (base 16).

The octal system uses the coefficients 0 through 7. Thus, the number 5467.42 can be either an octal number or a decimal number. Accordingly, if it is an octal number, it must be denoted as

$$(5467.42)_8$$

But if it is a decimal number, it must be denoted as

$$(5467.42)_{10}$$

The hexadecimal number system uses the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 and for the remaining six numbers uses the letters A, B, C, D, E, and F corresponding to the decimal numbers 10, 11, 12, 13, 14, and 15 respectively. Table 1.1 shows the first 16 numbers of the decimal, binary, octal, and hexadecimal systems.

TABLE 1.1 The first 16 decimal, binary, octal, and hexadecimal numbers.

Decimal (Base 10)	*Binary (Base 2)*	*Octal (Base 8)*	*Hexadecimal (Base 16)*
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

1.2 Binary, Octal, and Hexadecimal to Decimal Conversions

A number in base r other than *base 10*, can be converted to its decimal equivalent using the following steps:

1. Express the given number in the form of (1.1).
2. Add the terms following the rules of decimal addition.

Example 1.1

Convert the binary number $(1101.101)_2$ to its decimal equivalent.

Solution:

$$(1101.101)_2 = 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 + 1\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3}$$
$$= 8 + 4 + 0 + 1 + 0.5 + 0 + 0.125 = (13.625)_{10}$$

Example 1.2

Convert the octal number $(540.6)_8$ to its decimal equivalent.

Solution:

$$(540.6)_8 = 5\times 8^2 + 4\times 8^1 + 0\times 8^0 + 6\times 8^{-1}$$
$$= 5\times 64 + 4\times 8 + 0\times 1 + 6\times 8^{-1} = (352.75)_{10}$$

Example 1.3

Convert the hexadecimal number $(DB0.A)_{16}$ to its decimal equivalent.

Solution:

$$(DB0.A)_{16} = D\times 16^2 + B\times 16^1 + 0\times 16^0 + A\times 16^{-1}$$
$$= 13\times 256 + 11\times 16 + 0\times 1 + 10\times 16^{-1} = (3,504.625)_{10}$$

1.3 Decimal to Binary, Octal, and Hexadecimal Conversions

We have learned how to convert any number of any base other than base *10* to its equivalent decimal. Now we will learn how to convert a decimal number to another base number. The procedure is as follows:

- An integer decimal number can be converted to any other base, say r, by repeatedly dividing the given decimal number by r until the quotient becomes zero. The first remainder obtained becomes the least significant digit, and the last remainder becomes the most significant digit of the base r number.
- A fractional decimal number can be converted to any other base, say r, by repeatedly multiplying the given decimal number by r until a number with zero fractional part is obtained. This, however, may not be always possible, i.e., the conversion may be endless as some examples to follow will show.
- A mixed (integer and fractional) decimal number can be converted to any other base number, say r, by first converting the integer part, then converting the fractional part, and finally combining these two parts.

Example 1.4

Convert the decimal number $(39)_{10}$ to its binary equivalent.

Solution:

$$
\begin{aligned}
39/2 &= \text{Quotient } 19 + \text{Remainder } 1 \text{ (lsb)} \\
19/2 &= \text{Quotient } 9 + \text{Remainder } 1 \\
9/2 &= \text{Quotient } 4 + \text{Remainder } 1 \\
4/2 &= \text{Quotient } 2 + \text{Remainder } 0 \\
2/2 &= \text{Quotient } 1 + \text{Remainder } 0 \\
1/2 &= \text{Quotient } 0 + \text{Remainder } 1 \text{ (msb)}
\end{aligned}
$$

In the last step above, the quotient is 0; therefore, the conversion is completed and thus we have

$$(39)_{10} = (100111)_2$$

Example 1.5

Convert the decimal number $(0.39654)_{10}$ to its binary equivalent.

Solution:

$$
\begin{aligned}
0.39654 \times 2 &= 0.79308 = \mathbf{0} \text{ (msb of binary number)} + 0.79308 \\
0.79308 \times 2 &= 1.58616 = \mathbf{1} \text{ (next binary digit)} + 0.58616 \\
0.58616 \times 2 &= 1.17232 = \mathbf{1} + 0.17232 \\
0.17232 \times 2 &= 0.34464 = \mathbf{0} + 0.34464 \\
&\text{and so on}
\end{aligned}
$$

We observe that, for this example, the conversion is endless; this is because the given fractional

decimal number is not an exact sum of negative powers of *2*.

Therefore, for this example,

$$(0.39654)_{10} = (0.0110\ldots)_2$$

Example 1.6

Convert the decimal number $(0.84375)_{10}$ to its binary equivalent.

Solution:

$$\begin{aligned}
0.84375 \times 2 &= 1.6875 = \mathbf{1}\ (\text{msb of binary number}) + 0.6875\\
0.6875 \times 2 &= 1.375 = \mathbf{1}\ (\text{next binary digit}) + 0.375\\
0.375 \times 2 &= 0.75 = \mathbf{0} + 0.75\\
0.75 \times 2 &= 1.5 = \mathbf{1} + 0.5\\
0.5 \times 2 &= 1.0 = \mathbf{1}\ (\text{lsb}) + 0.0
\end{aligned}$$

Since the fractional part of the last step above is 0, the conversion is complete and thus

$$(0.84375)_{10} = (0.11011)_2$$

For this example, the conversion is exact; this is because

$$(0.84375)_{10} = (0.11011)_2 = 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-5}$$

Example 1.7

Convert the decimal number $(39.84375)_{10}$ to its binary equivalent.

Solution:

Here, we first convert the integer part, i.e., 39 to its equivalent binary, then we convert the fractional part to its equivalent binary, and finally we combine the two parts to form the entire binary number. Thus, from Example 1.4,

$$(39)_{10} = (100111)_2$$

and from Example 1.6,

$$(0.84375)_{10} = (0.11011)_2$$

Therefore,

$$(39.84375)_{10} = (100111.11011)_2$$

Conversion from *decimal–to–octal* is accomplished by repeated division by 8 for the integer part,

and by repeated multiplication by 8 for the fractional part.

Example 1.8

Convert the decimal number $(345.158)_{10}$ to its octal equivalent.

Solution:

We first convert the integer part, next the fractional part, and then we combine these.

Integer part conversion:

$$345/8 = \text{Quotient } 43 + \text{Remainder } 1 \text{ (lsb)}$$
$$43/8 = \text{Quotient } 5 + \text{Remainder } 3$$
$$5/8 = \text{Quotient } 0 + \text{Remainder } 5 \text{ (msb)}$$

Fractional part conversion:

$$0.158 \times 8 = 1.264 = \mathbf{1} \text{ (msb of fractional part)} + 0.264$$
$$0.264 \times 8 = 2.112 = \mathbf{2} \text{ (next octal digit)} + 0.112$$
$$\text{and so on}$$

We observe that the fractional part conversion is endless; therefore,

$$(345.158)_{10} = (531.12\ldots)_8$$

Conversion from *decimal–to–hexadecimal* is accomplished by repeated division by 16 for the integer part, and by repeated multiplication by 16 for the fractional part.

Example 1.9

Convert the decimal number $(389.125)_{10}$ to its hexadecimal equivalent.

Solution:

As before, we first convert the integer part, next the fractional part, and then we combine these.

Integer part conversion:

$$389/16 = \text{Quotient } 24 + \text{Remainder } 5 \text{ (lsb)}$$
$$24/16 = \text{Quotient } 1 + \text{Remainder } 8$$
$$1/16 = \text{Quotient } 0 + \text{Remainder } 1 \text{ (msb)}$$

Fractional part conversion:

$$0.125 \times 16 = 2.0 = \mathbf{2} \text{ (msb of fractional part)} + 0.0$$

We observe that the conversion of this example is exact; therefore,

$$(389.125)_{10} = (185.2)_{16}$$

1.4 Binary-Octal-Hexadecimal Conversions

Since $2^3 = 8$ and $2^4 = 16$, it follows that each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits. Accordingly, to perform binary–to–octal conversion, we partition the binary number into groups of three digits each starting from the binary point and proceeding to the left for the integer part and to the right of the binary point for the fractional part.

Example 1.10

Convert the binary number $(10110001101011.1111)_2$ to its octal equivalent.

Solution:

Since leading zeros (zeros to the left of the integer part of the number) and zeros added to the right of the last digit of the fractional part of the number do not alter the value of the number, we partition the number in groups of three digits by inserting a zero to the left of the number (i.e. a leading zero), and two zeros to the right of the given number, and then we assign the equivalent octal value to each group as shown below.

010	110	001	101	011	.	111	100
2	6	1	5	3		7	4

Therefore,

$$(10110001101011.1111)_2 = (26153.74)_8$$

Conversion from octal–to–binary is accomplished in the reverse procedure, i.e. each octal digit is converted to its binary equivalent as it is shown in the following example.

Example 1.11

Convert the octal number $(673.124)_8$ to its binary equivalent.

Solution:

Here, we replace each octal digit by its binary equivalent, i.e.,

$$(673.124)_8 = 110\ 111\ 011\ .\ 001\ 010\ 100$$
$$6\quad 7\quad 3\ .\ 1\quad 2\quad 4$$

Therefore,

$$(673.124)_8 = (110111011.001010100)_2$$

Conversion from binary–to–hexadecimal or hexadecimal–to–binary is performed similarly except that the binary number is divided into groups of four digits for the binary–to–hexadecimal conversion, or replacing each hexadecimal digit to its four digit binary equivalent in the hexadecimal–to–binary conversion.

Example 1.12

Convert the binary number $(10110001101011.111101)_2$ to its hexadecimal equivalent.

Solution:

For this example, we insert two leading zeros to the left of the integer part and two zeros to the right of the decimal part, we partition the given binary number in groups of four digits, and we assign the equivalent hexadecimal digit to each binary group, that is,

$$0010\ 1100\ 0110\ 1011\ .\ 1111\ 0100$$
$$2\quad C\quad 6\quad B\ .\ F\quad 4$$

Therefore,

$$(10110001101011.111101)_2 = (2C6B.F4)_{16}$$

Example 1.13

Convert the hexadecimal number $(306.D)_{16}$ to its binary equivalent.

Solution:

$$3\quad 0\quad 6\ .\ D$$
$$0011\ \ 0000\ \ 0110\ .\ 1101$$

Therefore,

$$(306.D)_{16} = (1100000110.1101)_2$$

1.5 Summary

- Any number may be represented by a series of coefficients as:

$$A_nA_{n-1}A_{n-2}\ldots\ldots A_2A_1A_0.A_{-1}A_{-2}\ldots\ldots A_{-n}$$

  In the familiar decimal number system, also referred to as has *base*-10 or *radix* 10, the A_k coefficients are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 and the subscript value k denotes the power of ten by which the coefficient must be multiplied.

- Digital computers use the binary (base 2) system which has only two coefficients, 0 and 1. In the binary system each coefficient A_k is multiplied by 2^k.

- In general, a number of base or radix r with coefficients A_k is expressed as

$$A_n\cdot r^n + A_{n-1}\cdot r^{n-1} + A_{n-2}\cdot r^{n-2} + \ldots + A_2\cdot r^2 + A_1\cdot r^1 + A_0\cdot r^0 + A_{-1}\cdot r^{-1} + \ldots + A_{-n}\cdot r^{-n}$$

- Two other numbers of interest are the octal (base 8) and hexadecimal (base 16). The octal system uses the coefficients 0 through 7. The hexadecimal number system uses the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 and for the remaining six numbers uses the letters A, B, C, D, E, and F corresponding to the decimal numbers 10, 11, 12, 13, 14, and 15 respectively.

- To convert a number in base r to its decimal equivalent we express the number in the coefficient-radix form given above and we add the terms following the rules of decimal addition.

- An integer decimal number can be converted to any other base, say r, by repeatedly dividing the given decimal number by r until the quotient becomes zero. The first remainder obtained becomes the least significant digit, and the last remainder becomes the most significant digit of the base r number.

- A fractional decimal number can be converted to any other base, say r, by repeatedly multiplying the given decimal number by r until a number with zero fractional part is obtained. This, however, may not be always possible, i.e., the conversion may be endless.

- A mixed (integer and fractional) decimal number can be converted to any other base number, say r, by first converting the integer part, then converting the fractional part, and finally combining these two parts.

- Conversion from decimal–to–octal is accomplished by repeated division by 8 for the integer part, and by repeated multiplication by 8 for the fractional part.

- Conversion from decimal–to–hexadecimal is accomplished by repeated division by 16 for the integer part, and by repeated multiplication by 16 for the fractional part.

- To perform binary–to–octal conversion, we partition the binary number into groups of three digits each starting from the binary point and proceeding to the left for the integer part and to the right of the binary point for the fractional part. Conversion from octal–to–binary is accomplished in the reverse procedure, i.e. each octal digit is converted to its binary equivalent.
- To perform binary–to–hexadecimal conversion, we partition the binary number into groups of four digits each starting from the binary point and proceeding to the left for the integer part and to the right of the binary point for the fractional part. Conversion from octal–to–binary is accomplished in the reverse procedure, i.e. each hexadecimal digit is converted to its binary equivalent.

1.6 Exercises

1. Convert the binary number $(11101.1011)_2$ to its decimal equivalent.

2. Convert the octal number $(651.7)_8$ to its decimal equivalent.

3. Convert the hexadecimal number $(EF9.B)_{16}$ to its decimal equivalent.

4. Convert the decimal number $(57)_{10}$ to its binary equivalent.

5. Convert the decimal number $(0.54379)_{10}$ to its binary equivalent.

6. Convert the decimal number $(0.79425)_{10}$ to its binary equivalent.

7.Convert the decimal number $(0.7890625)_{10}$ to its binary equivalent.

8. Convert the decimal number $(57.54379)_{10}$ to its binary equivalent.

9. Convert the decimal number $(543.815)_{10}$ to its octal equivalent.

10. Convert the decimal number $(683.275)_{10}$ to its hexadecimal equivalent.

11. Convert the binary number $(11011101111001.01111)_2$ to its octal equivalent.

12. Convert the octal number $(527.64)_8$ to its binary equivalent.

13. Convert the binary number $(1000110111001.01011)_2$ to its hexadecimal equivalent.

14. Convert the hexadecimal number $(A9C7.BD)_{16}$ to its binary equivalent.

1.7 Solutions to End-of-Chapter Exercises

Dear Reader:

The remaining pages on this chapter contain solutions to all end-of-chapter exercises.

You must, for your benefit, make an honest effort to solve these exercises without first looking at the solutions that follow. It is recommended that first you go through and solve those you feel that you know. For your solutions that you are uncertain, look over your procedures for inconsistencies and computational errors, review the chapter, and try again. Refer to the solutions as a last resort and rework those problems at a later date.

You should follow this practice with all end-of-chapter exercises in this book.

1.

$$(11101.1011)_2 = 1\times 2^4 + 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 + 1\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3} + 1\times 2^{-4}$$
$$= 16 + 8 + 4 + 0 + 1 + 0.5 + 0 + 0.125 + 0.0625 = (29.6875)_{10}$$

2.

$$((651.7)_8)_8 = 6\times 8^2 + 5\times 8^1 + 1\times 8^0 + 7\times 8^{-1}$$
$$= 6\times 64 + 5\times 8 + 1\times 1 + 7\times 8^{-1} = (425.875)_{10}$$

3.

$$(EF9.B)_{16} = E\times 16^2 + F\times 16^1 + 9\times 16^0 + B\times 16^{-1}$$
$$= 14\times 256 + 15\times 16 + 9\times 1 + 11\times 16^{-1} = (3,833.6875)_{10}$$

4.

$57/2$ = Quotient 28 + Remainder **1** (lsb)
$28/2$ = Quotient 14 + Remainder **0**
$14/2$ = Quotient 7 + Remainder **0**
$7/2$ = Quotient 3 + Remainder **1**
$3/2$ = Quotient 1 + Remainder **1**
$1/2$ = Quotient 0 + Remainder **1** (msb)

In the last step above, the quotient is 0; therefore, the conversion is completed and thus we have

$$(57)_{10} = (111001)_2$$

5.

$0.54379\times 2 = 1.08758 =$ **1** (msb of binary number) + 0.08758
$0.08758\times 2 = 0.17516 =$ **0** (next binary digit) + 0.17516
$0.17516\times 2 = 0.35032 =$ **0** + 0.35032
$0.35032\times 2 = 0.70064 =$ **0** + 0.70064
$0.70064\times 2 = 1.40128 =$ **1** + 0.40128
and so on

We observe that the conversion is endless; this is because the given fractional decimal number is not an exact sum of negative powers of *2*. Therefore, for this example,

$$(0.54379)_{10} = (0.10001\ldots)_2$$

6.

$$0.79425 \times 2 = 1.5885 = \mathbf{1}\ (\text{msb of binary number}) + 0.5885$$
$$0.5885 \times 2 = 1.177 = \mathbf{1}\ (\text{next binary digit}) + 0.177$$
$$0.177 \times 2 = 0.354 = \mathbf{0} + 0.354$$
$$0.354 \times 2 = 0.708 = \mathbf{0} + 0.708$$
$$0.708 \times 2 = 1.416 = \mathbf{1} + 0.416$$
$$0.416 \times 2 = 0.832 = \mathbf{0} + 0.832$$
and so on

We observe that the conversion is endless; this is because the given fractional decimal number is not an exact sum of negative powers of *2*. Therefore, for this example,

$$(0.79425)_{10} = (0.110010\ldots)_2$$

7.

$$0.7890625 \times 2 = 1.578125 = \mathbf{1}\ (\text{msb of binary number}) + 0.578125$$
$$0.578125 \times 2 = 1.15625 = \mathbf{1}\ (\text{next binary digit}) + 0.15625$$
$$0.15625 \times 2 = 0.3125 = \mathbf{0} + 0.3125$$
$$0.3125 \times 2 = 0.625 = \mathbf{0} + 0.625$$
$$0.625 \times 2 = 1.25 = \mathbf{1} + 0.25$$
$$0.25 \times 2 = 0.5 = \mathbf{0} + 0.5$$
$$0.5 \times 2 = 1.0 = \mathbf{1} + 0$$

Since the fractional part of the last step above is 0, the conversion is complete and thus

$$(0.7890625)_{10} = (0.1100101)_2$$

8.

From Exercise 4

$$(57)_{10} = (111001)_2$$

and from Exercise 5

$$(0.54379)_{10} = (0.10001\ldots)_2$$

Therefore,

$$(57.54379)_{10} = (111001.10001\ldots)_2$$

9.

Integer part conversion:

$$543/8 = \text{Quotient } 67 + \text{Remainder } \mathbf{7}\ (\text{lsb})$$
$$67/8 = \text{Quotient } 8 + \text{Remainder } \mathbf{3}$$
$$8/8 = \text{Quotient } 1 + \text{Remainder } \mathbf{0}$$
$$1/8 = \text{Quotient } 0 + \text{Remainder } \mathbf{1}\ (\text{msb})$$

Fractional part conversion:

$$0.815 \times 8 = 6.52 = \mathbf{6}\ (\text{msb of fractional part}) + 0.52$$
$$0.52 \times 8 = 4.16 = \mathbf{4}\ (\text{next octal digit}) + 0.16$$
$$\text{and so on}$$

We observe that the fractional part conversion is endless; therefore,

$$(543.815)_{10} = (1037.64\ldots)_8$$

10.

$$683/16 = \text{Quotient } 42 + \text{Remainder } 11 = \text{B (lsb)}$$
$$42/16 = \text{Quotient } 2 + \text{Remainder } 10 = \text{A}$$
$$2/16 = \text{Quotient } 0 + \text{Remainder } 2 \text{ (msb)}$$

Fractional part conversion:

$$0.275 \times 16 = 4.4 = 4\ (\text{msb of fractional part}) + 0.4$$
$$0.4 \times 16 = 6.4 = 6\ (\text{next hexadecimal digit}) + 0.4$$
$$\text{and so on}$$

We observe that the fractional part conversion is endless; therefore,

$$(683.275)_{10} = (2AB.46\ldots)_{16}$$

11.

$$\begin{array}{ccccccccc} 011 & 011 & 101 & 111 & 001 & . & 011 & 110 \\ 3 & 3 & 5 & 7 & 1 & & 3 & 6 \end{array}$$

Therefore,

$$(11011101111001.01111)_2 = (33571.36)_8$$

12.

$$(527.64)_8 = \begin{array}{ccccccc} 101 & 010 & 111 & . & 110 & 100 \\ 5 & 2 & 7 & . & 6 & 4 \end{array}$$

Therefore,

$$(527.64)_8 = (101010111.110100)_2$$

13.

$$\begin{array}{cccccccc} 0001 & 0001 & 1011 & 1001 & . & 0101 & 1000 \\ 1 & 1 & B & 9 & . & 5 & 8 \end{array}$$

Therefore,

$$(1000110111001.01011)_2 = (11B9.58)_{16}$$

14.

A	9	C	7	.	B	D
1010	1001	1100	0111	.	1011	1101

Therefore,

$$(A9C7.BD)_{16} = (1010100111000111.10111101)_2$$

Chapter 2

Operations in Binary, Octal, and Hexadecimal Systems

This chapter begins with an introduction to arithmetic operations in binary, octal, and hexadecimal numbers. The tens-complement and nines-complements in the decimal system and the twos-complement and ones-complements in the binary system are discussed.

2.1 Binary System Operations

In this section we will discuss binary addition. We will defer binary subtraction until we introduce the two's and one's complements in a later section of this chapter. Binary multiplication and binary division is discussed in Chapter 9 in conjunction with shift registers.

The addition of numbers in any numbering system is accomplished much the same manner as decimal addition, that is, the addition starts in the least significant position (right most position), and any carries are added in other positions to the left as it is done with the decimal system.

The binary number system employs the numbers *0* and *1* only; therefore, the possible combinations of binary addition are:

$$0+0=0 \qquad 0+1=1 \qquad 1+0=1 \qquad 1+1=0 \text{ with a carry of 1}$$

We observe that in a binary addition the largest digit in any position is *1*, just as in the decimal addition the largest digit is *9*. Furthermore, the above combinations indicate that if the number of ones to be added in any column is odd, the sum digit for that column will be *1*, and if the number of ones to be added in any column is even, then the sum digit for that column will be zero. Also, a carry occurs whenever there are two or more ones to be added in any column.

Example 2.1

Add the numbers $(101101101)_2$ and $(1110011)_2$

Solution:

$$\begin{array}{ll} 1111111 & \text{Carries} \\ 101101101 \\ \quad 1110011 & + \\ \hline 111100000 \end{array}$$

Check:

$$(101101101)_2 = (365)_{10}$$
$$(1110011)_2 = (115)_{10}$$

Then,

$$(111100000)_2 = (365 + 115)_{10} = (480)_{10}$$

Example 2.2

Add the numbers $(110110)_2$, $(101001)_2$, $(111000)_2$, $(10101)_2$, and $(100010)_2$

Solution:

```
   11
   11
  111111     Carries
  110110
  101001
  111000   +
   10101
  100010
 --------
 11001110
```

Check:

$$(110110)_2 = (54)_{10}$$
$$(101001)_2 = (41)_{10}$$
$$(111000)_2 = (56)_{10}$$
$$(10101)_2 = (21)_{10}$$
$$(100010)_2 = (34)_{10}$$

Then,

$$(11001110)_2 = (54 + 41 + 56 + 21 + 34)_{10} = (206)_{10}$$

2.2 Octal System Operations

The addition of octal numbers is also very similar to decimal numbers addition except that when the sum of two or more octal numbers exceeds seven, a carry occurs just as a carry occurs when

the sum of two or more decimal numbers exceeds nine. Table 2.1 summarizes the octal addition. This table can also be used for octal subtraction as it will be illustrated by Example 2.4.

TABLE 2.1 Table for Addition and subtraction of octal numbers

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	10
2	3	4	5	6	7	10	11
3	4	5	6	7	10	11	12
4	5	6	7	10	11	12	13
5	6	7	10	11	12	13	14
6	7	10	11	12	13	14	15
7	10	11	12	13	14	15	16

When Table 2.1 above is used for addition, we first locate the least significant digit of the first number (augend) in the upper row of the table, and then we locate the least significant digit of the second number (addend) in the left most column of the table. The intersection of the augend with the addend gives the sum of these two numbers. We follow the same procedure for all other digits from right to left.

Example 2.3

Add the numbers $(3527)_8$ and $(4167)_8$

Solution:

$$\begin{array}{l} 011 \quad \text{Carries} \\ \left.\begin{array}{l} 3527 \\ 4167 \end{array}\right\} + \\ \hline 7716 \end{array}$$

Starting with the least significant digit column above, we add 7 with 7 and the table gives us 16 i.e., 6 with a carry of 1. Next we add 6 and 2, with a carry of 1, or 6 and 3, and the table gives us 11 i.e., 1 with a carry of 1. Now we add 1, 5 and 1 (carry) and we get 7 with no carry. Finally, we add 4 and 3 which gives 7 and no carry. As before, we can check the sum for correctness by converting the numbers to their equivalent decimal numbers.

When Table 2.1 above is used for subtraction, we first find the least significant digit of the subtrahend (the smaller number) in the first row of the table. Then, in the same column, we locate the least significant digit of the minuend (the larger number). If the least significant digit of the minu-

end is less than the least significant digit of the subtrahend, a borrow occurs and from the numbers 10 through 16 in the table we choose the one whose least significant digit matches the least significant digit of the minuend. We find the difference by going across to the left most column. We follow the same procedure for all other digits from right to left.

We can use MATLAB conversion function **base2dec(s,b)** to convert the string number **s** of base **b** into its decimal (base 10) equivalent. **b** must be an integer between 2 and 36. For this example,

```
x=base2dec('3527',8); y=base2dec('4167',8); z=base2dec('7716',8); v=x+y; fprintf(' \n');...
fprintf('x=%5.0f \t',x); fprintf('y=%5.0f \t',y); fprintf('v=%5.0f \t',v);...
fprintf('z=%5.0f \t',z); fprintf(' \n')
```

```
x=1879   y=2167   v=4046   z=4046
```

Example 2.4

Subtract $(415)_8$ from $(614)_8$

Solution:

$$\left.\begin{matrix}614\\415\end{matrix}\right\}-$$
$$177$$

The least significant digit of the subtrahend is 5 and we locate it in the first row of Table 2.1. Going down that column where 5 appears, we choose 14 because the least significant digit of the minuend is 4. The difference, 7 in this case with a borrow, appears across to the left most column. Next, we add the borrow to the next digit of the subtrahend, 1 in this case, so now we must subtract 2 from 1. Continuing we locate 2 in the first row of the table and going down on the same column we choose 11 because the next digit of the minuend is 1, and again from the left most column we find that the difference is 7 with another borrow. Finally, we add that borrow to 4 and now we subtract 5 from 6 and this difference of 1 appears in the most significant position of the result with no borrow.

Check with MATLAB:

```
x=base2dec('614',8); y=base2dec('415',8); z=base2dec('177',8); v=x-y;...
fprintf(' \n');...
fprintf('x=%3.0f \t',x); fprintf('y=%3.0f \t',y); fprintf('v=%3.0f \t',v);...
fprintf('z=%3.0f \t',z); fprintf(' \n')
```

```
x=396   y=269   v=127   z=127
```

2.3 Hexadecimal System Operations

Hexadecimal addition and subtraction is accomplished similarly to that of addition and subtraction with octal numbers except that we use Table 2.2. When Table 2.2 below is used for addition, we first locate the least significant digit of the first number (augend) in the upper row of the table, and then we locate the least significant digit of the second number (addend) in the left most column of the table. The intersection of the augend with the addend gives the sum of these two numbers. We follow the same procedure for all other digits from right to left.

TABLE 2.2 Table for Addition and subtraction of hexadecimal numbers

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Example 2.5

Add the numbers $(F347)_{16}$ and $(E916)_{16}$

Solution:

$$\left.\begin{array}{r} F347 \\ E916 \end{array}\right\}+$$

$$1DC5D$$

Starting with the least significant digit column above, we add 7 with 6 and the table gives us D with no carry. Next, we add 4 and 1 and we get 5 with no carry. Now, we add 9 and 3 and we

get C with no carry. Finally, we add F and E and that gives 1D. As before, we can check the sum for correctness by converting the numbers to their equivalent decimal numbers.

Check with MATLAB:

```
x=base2dec('F347',16); y=base2dec('E916',16); z=base2dec('1DC5D',16); v=x+y;...
fprintf(' \n');...
fprintf('x=%6.0f \t',x); fprintf('y=%6.0f \t',y); fprintf('v=%6.0f \t',v);...
fprintf('z=%6.0f \t',z); fprintf(' \n')
```

```
x=62279   y=59670   v=121949   z=121949
```

Example 2.6

Subtract $(A9F8)_{16}$ from $(D5C7)_{16}$

Solution:

$$\begin{array}{r} \left.\begin{array}{r} D5C7 \\ A9F8 \end{array}\right\}- \\ \hline 2BCF \end{array}$$

The subtraction begins by locating the least significant digit of the subtrahend, 8 in this case, in the first row of Table 2.2, and going down the same column we find 17 on the last row of the table. Going across to the left most column we find that the difference is F with a borrow. Next, because of the borrow, we reduce the digit C of the minuend to B and from it we subtract F. The difference is found by locating F in the first row, going down the same column we find 1B, and going across to the left most column we find that the difference is C with a borrow. Now, because of the previous borrow, we reduce 5 to 4 and subtracting 9 from it we find that the difference is B with another borrow. Finally, because of the previous borrow, we reduce D to C and we subtract A from it. The difference is 2 with no borrow.

Check with MATLAB:

```
x=base2dec('D5C7',16); y=base2dec('A9F8',16); z=base2dec('2BCF',16); v=x–y;...
fprintf(' \n');...
fprintf('x=%3.0f \t',x); fprintf('y=%3.0f \t',y); fprintf('v=%3.0f \t',v);...
fprintf('z=%3.0f \t',z); fprintf(' \n')
```

```
x=54727   y=43512   v=11215   z=11215
```

2.4 Complements of Numbers

The subtraction operation is simplified by the use of the *complements of numbers*. For each *base-b* system there are two useful types of complements, the *b's-complement*, and the *(b-1)'s-complement*. Accordingly, for the base-10 system we have the tens-complements and the nines-complements,

for the base-2 we have the twos-complements and ones-complements, for the base-8 we have the eights-complements and sevens-complements, and for the base-16 we have the sixteens-complements and the fifteens-complements.

2.4.1 Tens-Complement

The tens-complement of a number can be found by subtracting the first non-zero least significant digit and all zeros to the right of it from 10; then, we subtract all other digits from 9.

Example 2.7

Find the tens-complement of 23567

Solution:

We first subtract 7 (lsd) from 10 and we get 3. This is the lsd of the tens-complement. For the remainder part of the tens-complement, we subtract 6, 5, 3, and 2 from 9 and we get 3, 4, 6, and 7 respectively. Therefore, the tens-complement of 23567 is 76433.

Example 2.8

Find the tens-complement of 0.8642

Solution:

We first subtract 2 (lsd) from 10 and all other digits from 9. Therefore, the tens-complement of 0.8642 is 0.1358.

Example 2.9

Find the tens-complement of 37.562

Solution:

We first subtract 2 (lsd) from 10 and all other digits from 9. Therefore, the tens-complement of 37.562 is 62.438.

2.4.2 Nines-Complement

The nines-complement of a number can be found by subtracting every digit (lsd) of that number from 9.

Example 2.10

Find the nines-complement of 23567

Solution:

We subtract every digit of the given number from 9 and we find that the nines-complement of 23567 is 76432. We observe that this complement is one less than 76433 which, as we found in Example 2.7, is the tens-complement of 23567. This is always the case, that is, the nines-complement is always one less than the tens-complement. Alternately, we can add 1 to the nines-complement to get the tens-complement.

Example 2.11

Find the nines-complement of 37.562

Solution:

We subtract every digit of the given number from 9 and we find that the nines-complement of 23567 is 76432.Therefore, the nines-complement of 37.562 is 62.437.

2.4.3 Twos-Complement

The twos-complement of a number can be found by leaving all the least significant zeros and the least significant one unchanged and then replacing all zeros with ones and all ones with zeros in all the other digits.

Example 2.12

Find the twos-complement of 1101100

Solution:

Starting from the right side of the given number we leave 100 unchanged and then for the remaining digits, i.e, 1101 we replace the ones with zeros and the zero with one. Therefore, the twos-complement of 1101100 is 0010100.

Example 2.13

Find the twos-complement of 0.1011

Solution:

We leave the lsd (last 1) unchanged and we replace the ones with zeros and the zero with one.

Therefore, the twos-complement of 0.1011 is 0.0101 . The leading 0 to the left of the binary point that separates the integer and fractional parts remains unchanged.

Example 2.14

Find the twos-complement of 1101100.1011

Solution:

We leave the lsd (last 1) unchanged and we replace the ones with zeros and the zeros with ones. Therefore, the twos-complement of 1101100.1011 is 0010011.0101 .

2.4.4 Ones-Complement

The ones-complement of a number can be found by replacing all zeros with ones and all ones with zeros.

Example 2.15

Find the ones-complement of 1101100

Solution:

Replacing all ones with zeros and all zeros with ones we find that the ones-complement of 1101100 is 0010011 . We observe that this complement is one less than 0010100 which, as we found in Example 2.12, is the twos-complement of 1101100 . This is always the case, that is, the ones- complement is always one less than the twos-complement. Alternately, we can add 1 to the ones- complement to get the twos-complement.

Example 2.16

Find the ones-complement of 0.1011

Solution:

Replacing all ones with zeros and all zeros with ones we find that the ones-complement of 0.1011 is 0.0100 . The leading 0 to the left of the binary point that separates the integer and fractional parts remains unchanged.

Example 2.17

Find the ones-complement of 1101100.1011

Solution:

Replacing all ones with zeros and all zeros with ones we find that the ones-complement of 1101100.1011 is 0010011.0100.

2.5 Subtraction with Tens- and Twos-Complements

We will assume that the numbers for the subtraction operation are both positive numbers. The subtraction operation using tens-complement or twos-complements is performed as follows:

1. Take the tens-complement or twos-complement of the subtrahend and add it to the minuend which remains unchanged.
2. Check the result (sum), and
   a. if an end carry occurs, discard it.
   b. if an end carry does not occur, take the tens-complement or twos-complement of the result (sum) and place a minus (–) sign in front of it.

Example 2.18

Perform the subtraction $(61435 - 02798)_{10}$ using the tens-complement method.

Solution:

$$\left.\begin{array}{r} \text{Minuend} = 61435 \text{ stays unchanged} \rightarrow 61435 \\ \text{Subtrahend} = 02798 \text{ take tens–complement} \rightarrow \underline{97202} \end{array}\right\} + \qquad \text{Discard end carry} \rightarrow 158637$$

Therefore, $(61435 - 02798)_{10} = (58637)_{10}$

Example 2.19

Perform the subtraction $(02798 - 61435)_{10}$ using the tens-complement method.

Solution:

$$\left.\begin{array}{r} \text{Minuend} = 02798 \text{ stays unchanged} \rightarrow 02798 \\ \text{Subtrahend} = 61435 \text{ take tens–complement} \rightarrow \underline{38565} \end{array}\right\} + \qquad \text{No end carry} \rightarrow 41363$$

Since there is no end carry, we take the tens-complement of the sum 41363 and we place a minus (–) sign in front of it resulting in -58637. Therefore, $(02798 - 61435)_{10} = (-58637)_{10}$.

Example 2.20

Perform the subtraction $(1101100 - 1011011)_2$ using the twos-complement method.

Solution:

$$\left.\begin{array}{r} \text{Minuend} = 1101100 \text{ stays unchanged} \rightarrow 1101100 \\ \text{Subtrahend} = 1011011 \text{ take twos–complement} \rightarrow \underline{0100101} \end{array}\right\} +$$

$$\text{Discard end carry} \rightarrow 10010001$$

Therefore, $(1101100 - 1011011)_2 = (0010001)_2$

Example 2.21

Perform the subtraction $(1011011 - 1101100)_2$ using the twos-complement method.

Solution:

$$\left.\begin{array}{r} \text{Minuend} = 1011011 \text{ stays unchanged} \rightarrow 1011011 \\ \text{Subtrahend} = 1101100 \text{ take twos–complement} \rightarrow \underline{0010100} \end{array}\right\} +$$

$$\text{No end carry} \rightarrow 1101100$$

Since there is no end carry, we take the twos-complement of the sum 1101100 and we place a minus (–) sign in front of it resulting in –0010001 .

Therefore, $(1011011 - 1101100)_2 = (-0010001)_2$.

2.6 Subtraction with Nines- and Ones-Complements

We will assume that the numbers for the subtraction operation are both positive numbers. The subtraction operation using nines-complement or ones-complements is performed as follows:

1. Take the nines-complement or ones-complement of the subtrahend and add it to the minuend which remains unchanged.

2. Check the result (sum), and

   a. if an end carry occurs, add 1 – referred to as *end around carry* – to the lsd of the result (sum).
   b. if an end carry does not occur, take the nines-complement or ones-complement of the result (sum) and place a minus (–) sign in front of it.

Example 2.22

Perform the subtraction $(61435 - 02798)_{10}$ using the nines-complement method.

Solution:

$$\left.\begin{array}{r}\text{Minuend} = 61435 \text{ stays unchanged} \rightarrow 61435 \\ \text{Subtrahend} = 02798 \text{ take nines–complement} \rightarrow \underline{97201}\end{array}\right\}+$$

$$\text{Make end carry an end around carry} \rightarrow 158636$$

$$\left.\begin{array}{r}58636 \\ \text{End around carry} \rightarrow \underline{1}\end{array}\right\}+$$

$$58637$$

Therefore, $(61435 - 02798)_{10} = (58637)_{10}$

Example 2.23

Perform the subtraction $(02798 - 61435)_{10}$ using the nines-complement method.

Solution:

$$\left.\begin{array}{r}\text{Minuend} = 02798 \text{ stays unchanged} \rightarrow 02798 \\ \text{Subtrahend} = 61435 \text{ take nines–complement} \rightarrow \underline{38564}\end{array}\right\}+$$

$$\text{No end carry} \rightarrow 41362$$

Since there is no end carry, we take the nines-complement of the sum 41362 and we place a minus (–) sign in front of it resulting in –58637. Therefore, $(02798 - 61435)_{10} = (-58637)_{10}$.

Example 2.24

Perform the subtraction $(1101100 - 1011011)_2$ using the ones-complement method.

Solution:

$$\left.\begin{array}{r}\text{Minuend} = 1101100 \text{ stays unchanged} \rightarrow 1101100 \\ \text{Subtrahend} = 1011011 \text{ take ones–complement} \rightarrow \underline{0100100}\end{array}\right\}+$$

$$\text{Make end carry an end around carry} \rightarrow 10010000$$

$$\left.\begin{array}{r}0010000 \\ \text{End around carry} \rightarrow \underline{1}\end{array}\right\}+$$

$$0010001$$

Therefore, $(1101100 - 1011011)_2 = (0010001)_2$.

Example 2.25

Perform the subtraction $(1011011 - 1101100)_2$ using the ones-complement method.

Solution:

$$\left.\begin{array}{r} \text{Minuend} = 1011011 \text{ stays unchanged} \rightarrow 1011011 \\ \text{Subtrahend} = 1101100 \text{ take ones–complement} \rightarrow \underline{0010011} \end{array}\right\} +$$

$$\text{No end carry} \rightarrow 1101110$$

Since there is no end carry, we take the ones-complement of the sum 1101110 and we place a minus (–) sign in front of it resulting in –0010001 .

Therefore, $(1011011 - 1101100)_2 = (-0010001)_2$.

More advanced topics on arithmetic operations and number representations will be discussed in Chapters 3 and 11.

2.7 Summary

- In a binary addition, if the number of ones to be added in any column is odd, the sum digit for that column will be *1*, and if the number of ones to be added in any column is even, then the sum digit for that column will be zero. Also, a carry occurs whenever there are two or more ones to be added in any column.
- The addition of octal numbers is also very similar to decimal numbers addition except that when the sum of two or more octal numbers exceeds seven, a carry occurs just as a carry occurs when the sum of two or more decimal numbers exceeds nine. Table 2.1 offers a convenient method for octal addition and subtraction.
- The addition and subtraction of hexadecimal numbers is conveniently performed with the use of Table 2.2.
- The subtraction operation is simplified by the use of the complements of numbers. For each *base-b* system there are two useful types of complements, the b's-complement, and the (b-1)'s-complement. Accordingly, for the base-10 system we have the tens-complements and the nines-complements, for the base-2 we have the twos-complements and ones-complements.
- The tens-complement of a number can be found by subtracting the least significant digit (lsd) of that number from 10 and then subtracting all other digits from 9.
- The nines-complement of a number can be found by subtracting every digit (lsd) of that number from 9.
- The twos-complement of a number can be found by leaving all the least significant zeros and the least significant one unchanged and then replacing all zeros with ones and all ones with zeros in all the other digits.
- The ones-complement of a number can be found by replacing all zeros with ones and all ones with zeros.
- The subtraction operation using tens-complement or twos-complements is performed as follows:
  1. Take the tens-complement or twos-complement of the subtrahend and add it to the minuend which remains unchanged.
  2. Check the result (sum), and
     a. if an end carry occurs, discard it.
     b. if an end carry does not occur, take the tens-complement or twos-complement of the result (sum) and place a minus (–) sign in front of it.

- The subtraction operation using nines-complement or ones-complements is performed as follows:

1. Take the nines-complement or ones-complement of the subtrahend and add it to the minuend which remains unchanged.
2. Check the result (sum), and
   a. if an end carry occurs, add 1 – referred to as end around carry – to the lsd of the result (sum).
   b. if an end carry does not occur, take the nines-complement or ones-complement of the result (sum) and place a minus (–) sign in front of it.

2.8 Exercises

1. Add the numbers $(110010010)_2$ and $(1011100)_2$

2. Add the numbers $(110110)_2$, $(101001)_2$, $(100111)_2$, $(11010)_2$, and $(111101)_2$

3. Add the numbers $(2735)_8$ and $(6741)_8$

4. Subtract $(145)_8$ from $(416)_8$

5. Add the numbers $(E743)_{16}$ and $(F9C8)_{16}$

6. Subtract $(8F9A)_{16}$ from $(C5D7)_{16}$

7. Construct a table similar to Tables 2.1 and 2.2 for addition and subtraction of binary numbers.

8. Subtract $(1011100)_2$ from $(110010010)_2$ using the table constructed in Exercise 7.

9. Find the tens-complement of 67235

10. Find the tens-complement of 0.4268

11. Find the tens-complement of 752.0368

12. Find the nines-complement of 67235

13. Find the nines-complement of 275.6083

14. Find the twos-complement of 1111110000

15. Find the twos-complement of 0.010100

16. Find the twos-complement of 1000010.0001

17. Find the ones-complement of 1000001

18. Find the ones-complement of 0.0101

19. Find the ones-complement of 101001.0100

20. Perform the subtraction $(43561 - 13820)_{10}$ using the tens-complement method.

21. Perform the subtraction $(13820 - 43561)_{10}$ using the tens-complement method.

22. Perform the subtraction $(1100100 - 1010011)_2$ using the twos-complement method.

23. Perform the subtraction $(1010011 - 1100100)_2$ using the twos-complement method.

24. Perform the subtraction $(43561 - 13820)_{10}$ using the nines-complement method.

25. Perform the subtraction $(13820 - 43561)_{10}$ using the nines-complement method.

26. Perform the subtraction $(1100100 - 1010011)_2$ using the ones-complement method.

27. Perform the subtraction $(1010011 - 1100100)_2$ using the ones-complement method.

28. A negative number is stored in a computing device in twos-complement form as

$$1100011.01011$$

What is the decimal value of this number?

29. The ones complement of a binary number N in a B-bit system is defined as $(2^B - N) - 1$. Prove that $-(-N) = N$

30. The twos complement of a binary number N in a B-bit system is defined as $2^B - N$. Using this definition prove that subtraction can be performed if we add the twos-complement of the subtrahend to the minuend.

2.9 Solutions to End-of-Chapter Exercises

1.

$$\begin{array}{rl} & 1 \quad \text{Carries} \\ \left.\begin{array}{r} 110010010 \\ 1011100 \end{array}\right\} & + \\ \hline 111101110 & \end{array}$$

Check:

$$(110010010)_2 = (402)_{10}$$
$$(1011100)_2 = (92)_{10}$$

Then,

$$(111101110)_2 = (402 + 92)_{10} = (494)_{10}$$

2.

$$\begin{array}{rl} & 11 \\ & 11111 \\ & 111111 \quad \text{Carries} \\ & 110110 \\ & 101001 \\ & 100111 \\ & 11010 \\ & 111101 \\ \hline 11011101 & \end{array} \quad +$$

Check:

$$(110110)_2 = (54)_{10}$$

$$(101001)_2 = (41)_{10}$$

$$(100111)_2 = (39)_{10}$$

$$(11010)_2 = (26)_{10}$$

$$(111101)_2 = (61)_{10}$$

Then,

$$(11001110)_2 = (54 + 41 + 39 + 26 + 34)_{10} = (221)_{10}$$

3.

$$\begin{array}{r} \text{11 Carries} \\ \left.\begin{array}{r} 2735 \\ 6741 \end{array}\right\} + \\ \hline 11676 \end{array}$$

Check with MATLAB:

```
x=base2dec('2735',8); y=base2dec('6741',8); z=base2dec('11676',8); v=x+y; fprintf(' \n');...
fprintf('x=%5.0f \t',x); fprintf('y=%5.0f \t',y); fprintf('v=%5.0f \t',v);...
fprintf('z=%5.0f \t',z); fprintf(' \n')
```

```
x= 1501   y= 3553   v= 5054   z= 5054
```

4.

$$\begin{array}{r} \left.\begin{array}{r} 416 \\ 145 \end{array}\right\} - \\ \hline 251 \end{array}$$

Check with MATLAB:

```
x=base2dec('416',8); y=base2dec('145',8); z=base2dec('251',8); v=x–y;...
fprintf(' \n');...
fprintf('x=%3.0f \t',x); fprintf('y=%3.0f \t',y); fprintf('v=%3.0f \t',v);...
fprintf('z=%3.0f \t',z); fprintf(' \n')
```

```
x=270   y=101   v=169   z=169
```

5.

$$\begin{array}{r} \left.\begin{array}{r} E743 \\ F9C8 \end{array}\right\} + \\ \hline 1E10B \end{array}$$

Check with MATLAB:

```
x=base2dec('E743',16); y=base2dec('F9C8',16); z=base2dec('1E10B',16); v=x+y;...
fprintf(' \n');...
fprintf('x=%6.0f \t',x); fprintf('y=%6.0f \t',y); fprintf('v=%6.0f \t',v);...
fprintf('z=%6.0f \t',z); fprintf(' \n')
```

```
x= 59203 y= 63944 v=123147 z=123147
```

6.

$$\left.\begin{matrix} C5D7 \\ 8F9A \end{matrix}\right\} - \\ \hline 363D$$

Check with MATLAB:

```
x=base2dec('C5D7',16); y=base2dec('8F9A',16); z=base2dec('363D',16); v=x–y;...
fprintf(' \n');...
fprintf('x=%3.0f \t',x); fprintf('y=%3.0f \t',y); fprintf('v=%3.0f \t',v);...
fprintf('z=%3.0f \t',z); fprintf(' \n')
```

```
x=50647   y=36762   v=13885   z=13885
```

7.

TABLE 2.3 Table for Addition and subtraction of binary numbers

0	1
0	1
1	10

When Table 2.3 above is used for addition, we first locate the least significant digit of the first number (augend) in the upper row of the table, and then we locate the least significant digit of the second number (addend) in the left most column of the table. The intersection of the augend with the addend gives the sum of these two numbers. We follow the same procedure for all other digits from right to left.

When Table 2.3 above is used for subtraction, we first find the least significant digit of the subtrahend (the smaller number) in the first row of the table. Then, in the same column, we locate the least significant digit of the minuend (the larger number). If the least significant digit of the minuend is less than the least significant digit of the subtrahend, a borrow occurs and from table we choose the one whose least significant digit matches the least significant digit of the minuend. We find the difference by going across to the left most column. We follow the same procedure for all other digits from right to left.

8.

$$\left.\begin{matrix} 110010010 \\ 1011100 \end{matrix}\right\} - \\ \hline 100110110$$

Check with MATLAB:

```
x=base2dec('110010010',2);  y=base2dec('1011100',2);  z=base2dec('100110110',2);...
v=x–y; fprintf(' \n');...
```

```
fprintf('x=%5.0f \t',x); fprintf('y=%5.0f \t',y); fprintf('v=%5.0f \t',v);...
fprintf('z=%5.0f \t',z); fprintf(' \n')
```

```
x =  402   y =  92   v =  310   z =  310
```

9.

We first subtract 5 (lsd) from 10 and we get 5. This is the lsd of the tens-complement. For the remainder part of the tens-complement, we subtract 3, 2, 7, and 6 from 9 and we get 6, 7, 2, and 3 respectively. Therefore, the tens-complement of 67235 is 32765.

10.

We first subtract 8 (lsd) from 10 and all other digits from 9. Therefore, the tens-complement of 0.4268 is 0.5732.

11.

We first subtract 8 (lsd) from 10 and all other digits from 9. Therefore, the tens-complement of 752.0368 is 247.9632.

12.

We subtract every digit of the given number from 9 and we find that the nines-complement of 67235 is 32764. We observe that this complement is one less than 32765 which we found in Exercise 9.

13.

We subtract every digit of the given number from 9 and we find that the nines-complement of 275.6083 is 724.3916.

14.

Starting from the right side of the given number we leave 10000 unchanged and then for the remaining digits, i.e, 11111 we replace the ones with zeros. Therefore, the twos-complement of 1111110000 is 0000010000 or simply 10000.

15.

We leave the last three digits (100) unchanged and we replace the zeros with ones and the one with zero. Therefore, the twos-complement of 0.010100 is 0.101100. The leading 0 to the left of the binary point that separates the integer and fractional parts remains unchanged.

16.

We leave the lsd (last 1) unchanged and we replace the ones with zeros and the zeros with ones. Therefore, the twos-complement of 1000010.0001 is 0111101.1111 or 111101.1111.

17.

Replacing all ones with zeros and all zeros with ones we find that the ones-complement of 1000001 is 0111110 or 111110.

18.

Replacing all ones with zeros and all zeros with ones we find that the ones-complement of 0.0101 is 0.1010. The leading 0 to the left of the binary point that separates the integer and fractional parts remains unchanged.

19.

Replacing all ones with zeros and all zeros with ones we find that the ones-complement of 101001.0100 is 010110.1011 or 10110.1011.

20. Here, the first non-zero least significant digit of the subtrahend is 2 and it is subtracted from 10. Thus,

$$\left.\begin{array}{r}\text{Minuend} = 43561 \text{ stays unchanged} \rightarrow 43561 \\ \text{Subtrahend} = 13820 \text{ take tens–complement} \rightarrow \underline{86180}\end{array}\right\}+$$

$$\text{Discard end carry} \rightarrow 129741$$

Therefore, $(43561 - 13820)_{10} = (29741)_{10}$

21.

$$\left.\begin{array}{r}\text{Minuend} = 13820 \text{ stays unchanged} \rightarrow 13820 \\ \text{Subtrahend} = 43561 \text{ take tens–complement} \rightarrow \underline{56439}\end{array}\right\}+$$

$$\text{No end carry} \rightarrow 70259$$

Since there is no end carry, we take the tens-complement of the sum 70259 and we place a minus (–) sign in front of it resulting in -29741. Therefore, $(13820 - 43561)_{10} = (-29741)_{10}$.

22.

$$\left.\begin{array}{r}\text{Minuend} = 1100100 \text{ stays unchanged} \rightarrow 1100100 \\ \text{Subtrahend} = 1010011 \text{ take twos–complement} \rightarrow \underline{0101101}\end{array}\right\}+$$

$$\text{Discard end carry} \rightarrow 10010001$$

Therefore, $(1100100 - 1010011)_2 = (10001)_2$

Check with MATLAB:

```
x=base2dec('1100100',2); y=base2dec('1010011',2); z=base2dec('0010001',2);...
```

```
v=x-y; fprintf(' \n');...
fprintf('x=%5.0f \t',x); fprintf('y=%5.0f \t',y); fprintf('v=%5.0f \t',v);...
fprintf('z=%5.0f \t',z); fprintf(' \n')

x =  100     y =  83     v =  17     z =  17
```

23.

$$\left.\begin{array}{r}\text{Minuend} = 1010011 \text{ stays unchanged} \rightarrow 1010011 \\ \text{Subtrahend} = 1100100 \text{ take twos-complement} \rightarrow \underline{0011100}\end{array}\right\}+$$

$$\text{No end carry} \rightarrow 1101111$$

Since there is no end carry, we take the twos-complement of the sum 1101111 and we place a minus (–) sign in front of it resulting in -10001.

Therefore, $(1010011 - 1100100)_2 = (-10001)_2$.

24.

$$\left.\begin{array}{r}\text{Minuend} = 43561 \text{ stays unchanged} \rightarrow 43561 \\ \text{Subtrahend} = 13820 \text{ take nines-complement} \rightarrow \underline{86179}\end{array}\right\}+$$

$$\text{Make end carry an end around carry} \rightarrow 129740$$

$$\left.\begin{array}{r}29740 \\ \underline{\text{End around carry} \rightarrow 1}\end{array}\right\}+$$

$$29741$$

Therefore, $(43561 - 13820)_{10} = (29741)_{10}$

25.

$$\left.\begin{array}{r}\text{Minuend} = 13820 \text{ stays unchanged} \rightarrow 13820 \\ \text{Subtrahend} = 43561 \text{ take nines-complement} \rightarrow \underline{56438}\end{array}\right\}+$$

$$\text{No end carry} \rightarrow 70258$$

Since there is no end carry, we take the nines-complement of the sum 70258 and we place a minus (–) sign in front of it resulting in -29741.

Therefore, $(13820 - 43561)_{10} = (-29741)_{10}$.

26.

$$\left.\begin{array}{r}\text{Minuend} = 1100100 \text{ stays unchanged} \rightarrow 1100100 \\ \text{Subtrahend} = 1010011 \text{ take ones–complement} \rightarrow \underline{0101100}\end{array}\right\} +$$

$$\text{Make end carry an end around carry} \rightarrow 10010000$$

$$\left.\begin{array}{r}0010000 \\ \underline{\text{End around carry} \rightarrow 1}\end{array}\right\} +$$

$$0010001$$

Therefore, $(1100100 - 1010011)_2 = (10001)_2$.

27.

$$\left.\begin{array}{r}\text{Minuend} = 1010011 \text{ stays unchanged} \rightarrow 1010011 \\ \text{Subtrahend} = 1100100 \text{ take ones–complement} \rightarrow \underline{0011011}\end{array}\right\} +$$

$$\text{No end carry} \rightarrow 1101110$$

Since there is no end carry, we take the ones-complement of the sum 1101110 and we place a minus (–) sign in front of it resulting in –0010001.

Therefore, $(1011011 - 1101100)_2 = (-10001)_2$.

28.

$$1100011.01011 \text{ twos-complement} \rightarrow 0011100.10101$$

and the unsigned decimal value is

$$(0011100.10101)_2 \rightarrow \left(16 + 8 + 4 + \frac{1}{2} + \frac{1}{8} + \frac{1}{32}\right)_{10} = (28.65625)_{10}$$

It is known that the given number is negative; therefore, its true decimal value is –28.65625.

29.

Adding and subtracting $(2^B - 1)$ to $-(-N)$ we get

$$-(-N) = (2^B - 1) - [(2^B - 1) - N] = (2^B - 1) - (2^B - 1) + N = N$$

30.

Suppose that X and Y are two positive numbers where X is the minuend and Y is the subtrahend. Then,

$$X + (-Y) = X + (2^B - Y) = 2^B - (Y - X) \quad (1)$$

but $X + (2^B - Y) = X + (-Y)$ or $2^B = 0$. Then, (1) above reduces to

$$X + (-Y) = X + (2^B - Y) = 2^B - (Y - X) = 0 - (Y - X) = X - Y$$

Chapter 3

Sign Magnitude and Floating Point Arithmetic

This chapter begins with an introduction to sign magnitude representation of binary numbers. We discuss floating point arithmetic for representing large numbers and the IEEE standard that specifies single precision (32 bit) and double precision (64 bit) floating point representation of numbers.

3.1 Signed Magnitude of Binary Numbers

In a sign magnitude representation, the most significant digit (bit) is 0 for positive binary numbers and 1 for negative binary numbers. As we know, for large binary numbers it is more convenient to use hexadecimal numbers. In most computers, negative binary numbers are stored in twos-complement form.

Example 3.1

Convert $(+32749)_{10}$ and $(-32749)_{10}$ into sign magnitude binary representation form.

Solution:

The conversions will result in large binary numbers and so it may be convenient to perform a decimal-to-hexadecimal conversion procedure as presented in Chapter 1 where we can find the hexadecimal equivalent by repeated division by 16. However, we will use the MATLAB conversion function **dec2hex(d)**. This function converts the decimal integer **d** to a hexadecimal string with the restriction that **d** is a non-negative integer smaller than 2^{52}.

For $(+32749)_{10}$, **dec2hex(32749)** returns $(7FED)_{16} = (0111\ 1111\ 1110\ 1101)_2$ where the spaces were added to indicate the hexadecimal groupings. We observe that the most significant digit is 0 indicating that this is a positive binary number. Then,

$$(+32749)_{10} = (0111\ 1111\ 1110\ 1101)_2$$

For $(-32749)_{10}$, in binary sign form must have 1 for its most significant digit and it is represented in twos-complement form which is obtained by forming the twos-complement of $(0111\ 1111\ 1110\ 1101)_2$ resulting in $(1000\ 0000\ 0001\ 0011)_2$. Then,

$$(-32749)_{10} = (1000\ 0000\ 0001\ 0011)_2$$

Example 3.2

Convert the decimal numbers $(+57823)_{10}$ and $(-57823)_{10}$ into sign magnitude binary representation form.

Solution:

As in the previous example, we will use the MATLAB conversion function **dec2hex(d)** to convert the decimal integer **d** to a hexadecimal string.

For $(+57823)_{10}$, **dec2hex(57823)** returns $(E1DF)_{16} = (1110\ 0001\ 1101\ 1111)_2$ where the spaces were added to indicate the hexadecimal groupings. We observe that the most significant digit is 1 indicating that this is a negative binary number, when in fact it is not a negative number. The problem here is that we have overlooked the fact that in sign magnitude binary representation form in an 8-bit binary number the equivalent decimal numbers range from $(-128)_{10}$ to $(+127)_{10}$, in a 16-bit binary number the equivalent decimal numbers range from $(-32768)_{10}$ to $(+32767)_{10}$, in a 32-bit binary number the equivalent decimal numbers range from $(-2, 147, 483, 648)_{10}$ to $(+2, 147, 483, 647)_{10}$, and so on.

Therefore, we cannot represent the numbers $(+57823)_{10}$ and $(-57823)_{10}$ into sign magnitude binary representation form with 16 bits; we must represent them with 32 bits. Hence,

$$(+57823)_{10} = (0000\ 0000\ 0000\ 0000\ 1110\ 0001\ 1101\ 1111)_2$$

$$(-57823)_{10} = (1111\ 1111\ 1111\ 1111\ 0001\ 1110\ 0010\ 0001)_2$$

3.2 Floating Point Arithmetic

With our everyday arithmetic operations, such as with dollars and cents, the decimal point that separates the fractional part from the integer part is fixed. However, digital computers use *floating point arithmetic* where the binary point, i.e., the point that separates the fractional part from the integer part in a binary number, is floating.

Floating point arithmetic is based on scientific notation, for instance,

$$1,010,000 = 1.01 \times 10^6$$

where 1.01 is referred to as the *mantissa* or *fraction* and 6 is the *exponent* or *characteristic*.

A floating point number representation is said to be *normalized* if the mantissa is within the range

$$1/\text{base} \le \text{mantissa} < 1 \tag{3.1}$$

Thus, the normalized floating point representation of $1,010,000$ is 0.101×10^5.
We should remember that the number 0 has an *unnormalized* value because it has no non-zero digit to be placed just to the right of decimal or binary point.

Henceforth, we will denote the exponent with the letter E and the mantissa (fraction) with the letter F. Accordingly, a binary number in a register can be represented as in Figure 3.1.

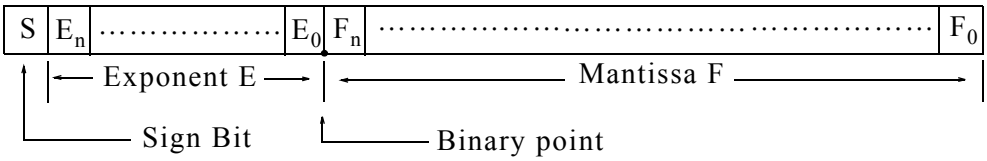


Figure 3.1. Typical representation of a binary number in floating point arithmetic

Our subsequent discussion will be restricted to the IEEE (Institute of Electrical and Electronics Engineers) *Standard 754* for floating point arithmetic which specifies the *single precision* (32 bit) and double precision (64 bit) floating point arithmetic.

3.2.1 The IEEE Single Precision Floating Point Arithmetic

The IEEE single precision floating point representation uses a 32-bit word consisting of the sign bit denoted as S, 8 bits for the exponent, denoted as E_7 through E_0, and 23 bits for the fraction, denoted as F_{22} through F_0 as shown in Figure 3.2 where the exponent E and the fraction F are always interpreted as positive base-2 numbers; they are not twos-complement numbers.

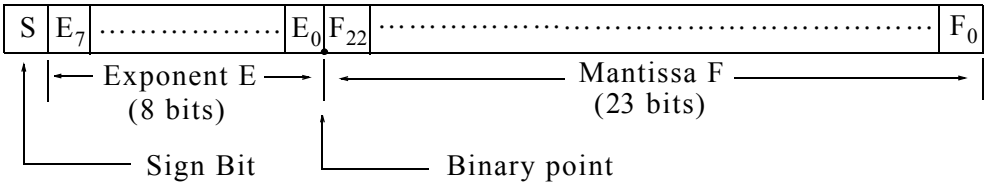


Figure 3.2. The IEEE single precision floating point arithmetic

The sign bit is 0 for positive numbers and 1 for negative numbers. Then, for different values of the exponent E, and the fraction F, we get the following possibilities:

1. If $S = 0$, $E = 0$, and $F = 0$, then $Value = +0$. In other words,

$$\underbrace{0}_{S} \quad \underbrace{00000000}_{E} \quad \underbrace{00000000000000000000}_{F} \qquad Value = +0$$

2. If $S = 1$, $E = 0$, and $F = 0$, then $Value = -0$. In other words,

$$\underbrace{1}_{S} \quad \underbrace{00000000}_{E} \quad \underbrace{00000000000000000000}_{F} \qquad \text{Value} = -0$$

3. If $S = 0$, $E = 255$, and $F = 0$, then Value $= +\infty$. In other words,

$$\underbrace{0}_{S} \quad \underbrace{11111111}_{E} \quad \underbrace{00000000000000000000}_{F} \qquad \text{Value} = +\text{Infinity}$$

4. If $S = 1$, $E = 255$, and $F = 0$, then Value $= -\infty$. In other words,

$$\underbrace{1}_{S} \quad \underbrace{11111111}_{E} \quad \underbrace{00000000000000000000}_{F} \qquad \text{Value} = -\text{Infinity}$$

5. If $S = 0$ or $S = 1$, $E = 255$, and $F \neq 0$, then Value = NaN*. In other words,

$$\underbrace{0 \text{ or } 1}_{S} \quad \underbrace{11111111}_{E} \quad \underbrace{00100111000000000000}_{F} \qquad \text{Value} = \text{NaN}$$

6. If $S = 0$ or $S = 1$, $0 < E < 255$, and $F = 0$ or $F \neq 0$, then

$$(\text{Value})_{10} = (-1)^{S} \cdot 2^{(E-127)} \cdot (1.F) \tag{3.2}$$

where the factor 1.F is included to imply when the relation of (3.2) is used, it is understood, although not shown, that the mantissa (fraction) is prefixed with an implicit leading 1 followed by the binary point. Thus, in accordance with (3.2), the representation

$$\underbrace{0}_{S = +} \quad \underbrace{10000010}_{E = (130)_{10}} \quad \underbrace{10100000000000000000}_{F = (1.101)_2}$$

has the decimal value

$$(\text{Value})_{10} = (-1)^{0} \cdot 2^{(130-127)} \cdot (1.101)_2 = 1 \times 2^{3} \times (1 + 0.5 + 0.125)_{10} = (13)_{10}$$

Obviously, the negative value of this, that is, Value $= -13$, is represented as

$$\underbrace{1}_{S = -} \quad \underbrace{10000010}_{E = (130)_{10}} \quad \underbrace{10100000000000000000}_{F = (1.101)_2}$$

7. If $S = 0$ or $S = 1$, $E = 0$, and $F \neq 0$, then,

* *NaN = Not a Number. It occurs in certain operations such as 0/0, ∞/∞, etc.*

$$(\text{Value})_{10} = (-1)^{S} \cdot 2^{(-126)} \cdot (0.F) \tag{3.3}$$

where the factor $0.F$ is included to imply when the relation of (3.3) is used, it is understood, although not shown, that the mantissa (fraction) is prefixed with an implicit leading 0 followed by the binary point. Thus, in accordance with (3.3), the representation

$$\underbrace{0}_{S = +} \qquad \underbrace{00000000}_{E = 0} \qquad \underbrace{10100000000000000000}_{F = (0.101)_2}$$

has the decimal value

$$(\text{Value})_{10} = (-1)^{0} \cdot 2^{(-126)} \cdot (0.101)_2 = 1 \times 2^{(-126)} \times (0 + 0.5 + 0.125)_{10} = (7.3468 \times 10^{-39})_{10}$$

Obviously, relation (3.3) is used with very small numbers.

Example 3.3

Give the IEEE single precision floating point representation for the number $(704)_{10}$

Solution:

The given number lies between $512 = 2^9$ and $1024 = 2^{10}$ and since we want the mantissa (fraction) to be just greater than 1 to conform to the $1.F$ factor in (3.2), we divide $(704)_{10}$ by $(512)_{10}$ which is the smallest power of base-2 that will convert the given number to a mixed number with integer part 1 and a decimal fractional part. Then,

$$704/512 = 1.375 = 1 + 0.25 + 0.125 = 1 + 2^{-2} + 2^{-3} = (1.011)_2$$

Since we divided the given number by $512 = 2^9$, we must have

$$2^{(E-127)} = 2^9$$

from which

$$E - 127 = 9$$

or

$$E = (136)_{10} = (10001000)_2$$

Thus, the number $(704)_{10}$ in IEEE single precision floating point representation is

$$\underbrace{0}_{S = +} \qquad \underbrace{10001000}_{E = 136} \qquad \underbrace{10101000000000000000}_{F = (1.011)_2}$$

Check:

$$(\text{Value})_{10} = (-1)^0 \cdot 2^{(136-127)} \cdot (1.011)_2 = 1 \times 2^9 \times (1 + 0.25 + 0.125) = 704$$

Example 3.4

Give the IEEE single precision floating point representation for the number $(0.6875)_{10}$

Solution:

The given number lies between $0.5 = 2^{-1}$ and $1 = 2^0$ and since we want the mantissa (fraction) to be just larger than 1 to conform to the 1.F factor in (3.2), we divide $(0.6875)_{10}$ by $(0.5)_{10}$ which is the smallest power of base-2 that will convert the given number to mixed number with integer part 1 and a decimal fractional part. Then,

$$0.6875/0.5 = 1.375 = 1 + 0.25 + 0.125 = 1 + 2^{-2} + 2^{-3} = (1.011)_2$$

Since we divided the given number by $0.5 = 2^{-1}$, we must have

$$2^{(E-127)} = 2^{-1}$$

from which

$$E - 127 = -1$$

or

$$E = (126)_{10} = (01111110)_2$$

Thus, the number $(0.6875)_{10}$ in IEEE single precision floating point representation is

$$\underbrace{0}_{S = +} \qquad \underbrace{01111110}_{E = (126)_{10}} \qquad \underbrace{10101000000000000000}_{F = (1.011)_2}$$

Check:

$$(\text{Value})_{10} = (-1)^0 \cdot 2^{(126-127)} \cdot (1.011)_2 = 1 \times 2^{-1} \times (1 + 0.25 + 0.125) = (0.6875)_{10}$$

Example 3.5

Give the IEEE single precision floating point representation for the smallest negative number.

Solution:

From Figure 3.2, the mantissa (fraction) all bits in the F_{22} through F_1 and $F_0 = 1 \rightarrow 2^{(-23)}$. From (3.3), $(\text{Value})_{10} = (-1)^S \cdot 2^{(-126)} \cdot (0.F)$. Then,

$$\text{Smallest negative value} = (-1)^1 \cdot 2^{(-126)} \cdot 2^{(-23)} = -2^{(-149)} = (1.4013 \times 10^{-45})_{10}$$

3.2.2 The IEEE Double Precision Floating Point Arithmetic

The IEEE *double precision* floating point representation uses a 64-bit word consisting of the sign bit denoted as S, 11 bits for the exponent, denoted as E_{10} through E_0, and 52 bits for the fraction, denoted as F_{51} through F_0 as shown in Figure 3.3 where the exponent E and the fraction F are always interpreted as positive base-2 numbers; they are not twos-complement numbers.

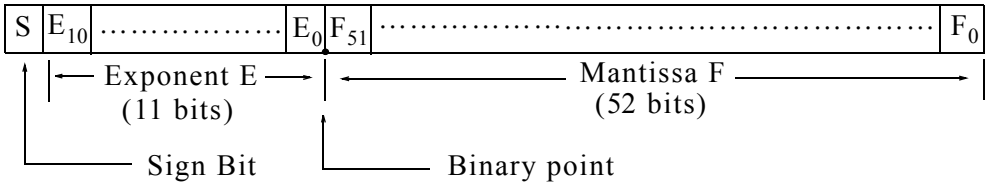


Figure 3.3. The IEEE double precision floating point arithmetic

The sign bit is 0 for positive numbers and 1 for negative numbers. Then, for different values of the exponent E, and the fraction F, we get the following possibilities.

1. If S = 0, E = 0, and F = 0, then Value = +0. In other words,

$$\underbrace{0}_{S}\quad\underbrace{00000000000}_{E}\quad\underbrace{00\ldots\ldots\ldots\ldots\ldots0}_{F\ (52\ bits\ long)}\qquad Value = +0$$

2. If S = 1, E = 0, and F = 0, then Value = –0. In other words,

$$\underbrace{1}_{S}\quad\underbrace{00000000000}_{E}\quad\underbrace{00\ldots\ldots\ldots\ldots\ldots0}_{F\ (52\ bits\ long)}\qquad Value = -0$$

3. If S = 0, E = 2047, and F = 0, then Value = +∞. In other words,

$$\underbrace{0}_{S}\quad\underbrace{11111111111}_{E}\quad\underbrace{00\ldots\ldots\ldots\ldots\ldots0}_{F\ (52\ bits\ long)}\qquad Value = +Infinity$$

4. If S = 1, E = 2047, and F = 0, then Value = –∞. In other words,

$$\underbrace{1}_{S}\quad\underbrace{11111111111}_{E}\quad\underbrace{00\ldots\ldots\ldots\ldots\ldots0}_{F\ (52\ bits\ long)}\qquad Value = -Infinity$$

5. If S = 0 or S = 1, E = 2047, and $F \neq 0$, then Value = NaN. In other words,

$$\underbrace{0\ or\ 1}_{S}\quad\underbrace{11111111111}_{E}\quad\underbrace{0010011100\ldots\ldots0}_{F\ (52\ bits\ long)}\qquad Value = NaN$$

6. If $S = 0$ or $S = 1$, $0 < E < 2047$, and $F = 0$ or $F \neq 0$, then

$$\text{Value} = (-1)^S \cdot 2^{(E-1022)} \cdot (1.F) \tag{3.4}$$

where the factor $1.F$ is included to imply when the relation of (3.4) is used, it is understood, although not shown, that the mantissa (fraction) is prefixed with an implicit leading 1 followed by the binary point. Thus, in accordance with (3.4), the representation

$$\underbrace{0}_{S = +} \qquad \underbrace{10000010000}_{E = 1040} \qquad \underbrace{10100\ldots\ldots\ldots\ldots 0}_{F = 1.101}$$

has the decimal value

$$\text{Value} = (-1)^0 \cdot 2^{(1040-1022)} \cdot (1.101) = 1 \times 2^{18} \times 1.101 = (2.8862 \times 10^5)_{10}$$

7. If $S = 0$ or $S = 1$, $E = 0$, and $F \neq 0$, then,

$$\text{Value} = (-1)^S \cdot 2^{(-1022)} \cdot (0.F) \tag{3.5}$$

where the factor $0.F$ is included to imply when the relation of (3.5) is used, it is understood, although not shown, that the mantissa (fraction) is prefixed with an implicit leading 0 followed by the binary point. Thus, in accordance with (3.5), the representation

$$\underbrace{0}_{S = +} \qquad \underbrace{00000000000}_{E = 0} \qquad \underbrace{10100\ldots\ldots\ldots\ldots 0}_{F = 0.101}$$

has the decimal value

$$\text{Value} = (-1)^0 \cdot 2^{(-1022)} \cdot (0.101)_2 = 1 \times 2^{(-1022)} \times (0 + 0.5 + 0.125)_{10} = (1.3907 \times 10^{-308})_{10}$$

3.3 Summary

- In a sign magnitude representation, the most significant digit (bit) is 0 for positive binary numbers and 1 for negative binary numbers.
- In most computers, negative binary numbers are stored in twos-complement form.
- It is very convenient to use the MATLAB conversion function **dec2hex(d)** which converts the decimal integer **d** to a hexadecimal string with the restriction that **d** is a non-negative integer smaller than 2^{52}.
- In sign magnitude binary representation form in an 8-bit binary number the equivalent decimal numbers range from $(-128)_{10}$ to $(+127)_{10}$, in a 16-bit binary number the equivalent decimal numbers range from $(-32768)_{10}$ to $(+32767)_{10}$, in a 32-bit binary number the equivalent decimal numbers range from $(-2, 147, 483, 648)_{10}$ to $(+2, 147, 483, 647)_{10}$, and so on.
- Digital computers use floating point arithmetic where the binary point, i.e., the point that separates the fractional part from the integer part in a binary number, is floating. Floating point arithmetic is based on scientific notation where a number consists of a mantissa and an exponent.
- A floating point number representation is said to be normalized if the mantissa is within the range

$$1/\text{base} \le \text{mantissa} < 1$$

- A number 0 has an *unnormalized* value because it has no non-zero digit to be placed just to the right of decimal or binary point.
- The IEEE single precision floating point representation uses a 32-bit word consisting of the sign bit denoted as S, 8 bits for the exponent, denoted as E_7 through E_0, and 23 bits for the fraction, denoted as F_{22} through F_0. The exponent E and the fraction F are always interpreted as positive base-2 numbers; they are not twos-complement numbers. The sign bit is 0 for positive numbers and 1 for negative numbers.
- The IEEE double precision floating point representation uses a 64-bit word consisting of the sign bit denoted as S, 11 bits for the exponent, denoted as E_{10} through E_0, and 52 bits for the fraction, denoted as F_{51} through F_0. The exponent E and the fraction F are always interpreted as positive base-2 numbers; they are not twos-complement numbers. The sign bit is 0 for positive numbers and 1 for negative numbers.

3.4 Exercises

1. Convert $(+74329)_{10}$ and $(-74329)_{10}$ into sign magnitude binary representation form.

2. Convert the decimal numbers $(+47387)_{10}$ and $(-47387)_{10}$ into sign magnitude binary representation form.

3. Give the IEEE single precision floating point representation for the number $(1253)_{10}$

4. Give the IEEE single precision floating point representation for the number $(-1253)_{10}$

5. Give the IEEE single precision floating point representation for the number $(0.485)_{10}$

6. Give the IEEE single precision floating point representation for the number $(-0.485)_{10}$

3.5 Solutions to-End-of-Chapter Exercises

1.

For $(+74329)_{10}$, **dec2hex(74329)** returns $(12259)_{16} = (0001\ 0010\ 0010\ 0101\ 1001)_2$ where the spaces were added to indicate the hexadecimal groupings. We observe that the most significant digit is 0 indicating that this is a positive binary number. Thus,

$$(+74329)_{10} = (0001\ 0010\ 0010\ 0101\ 1001)_2$$

For $(-32749)_{10}$, in binary sign form must have 1 for its most significant digit and it is represented in twos-complement form which is obtained by forming the twos-complement of $(0001\ 0010\ 0010\ 0101\ 1001)_2$ resulting in $(1110\ 1101\ 1101\ 1010\ 0111)_2$. Thus,

$$(-32749)_{10} = (1110\ 1101\ 1101\ 1010\ 0111)_2$$

2.

For $(+47387)_{10}$, **dec2hex(47387)** returns $(B91B)_{16} = (1011\ 1001\ 0001\ 1011)_2$ where the spaces were added to indicate the hexadecimal groupings. We observe that the most significant digit is 1 indicating that this is a negative binary number, which is not. The problem here is that in a 16-bit binary number the equivalent decimal numbers range from $(-32768)_{10}$ to $(+32767)_{10}$. Therefore, we must represent the given numbers with 32 bits. Hence,

$$(+47387)_{10} = (0000\ 0000\ 0000\ 0000\ 1011\ 1001\ 0001\ 1011)_2$$

$$(-47387)_{10} = (1111\ 1111\ 1111\ 1111\ 0100\ 0110\ 1110\ 0101)_2$$

3.

The given number lies between $1024 = 2^{10}$ and $2048 = 2^{11}$ and since we want the mantissa (fraction) to be just greater than 1 to conform to the 1.F factor in (3.2), we divide $(1253)_{10}$ by $(1024)_{10}$ which is the smallest power of base-2 that will convert the given number to a mixed number with integer part 1 and a decimal fractional part. Then,

$$1253/1024 = 1.2236 = 1 + 0.125 + 0.0625 + 0.03125 + \ldots$$
$$= 1 + 2^{-3} + 2^{-4} + 2^{-5} + \ldots = (1.00111\ldots)_2$$

Since we divided the given number by $1024 = 2^{10}$, we must have

$$2^{(E-127)} = 2^{10}$$

from which

$$E - 127 = 10$$

or

$$E = (137)_{10} = (10001001)_2$$

Thus, the number $(1253)_{10}$ in IEEE single precision floating point representation is

$$\underbrace{0}_{S = +} \qquad \underbrace{10001001}_{E = 137} \qquad \underbrace{100111\ldots\ldots\ldots\ldots}_{F = (1.00111\ldots)_2}$$

Check:

$$(\text{Value})_{10} = (-1)^0 \cdot 2^{(137-127)} \cdot (1.00111\ldots)_2$$
$$= 1 \times 2^{10} \times (1 + 0.125 + 0.0625 + 0.03125 + \ldots) = 1253$$

4.

The results are the same as in Exercise 3 except that the sign bit is one, and thus the number $(-1253)_{10}$ in IEEE single precision floating point representation is

$$\underbrace{1}_{S = -} \qquad \underbrace{10001001}_{E = 137} \qquad \underbrace{100111\ldots\ldots\ldots\ldots}_{F = (1.00111\ldots)_2}$$

5.

The given number lies between $0.25 = 2^{-2}$ and $0.5 = 2^{-1}$ and since we want the mantissa (fraction) to be just larger than 1 to conform to the 1.F factor in (3.2), we divide $(0.485)_{10}$ by $(0.25)_{10}$ which is the smallest power of base-2 that will convert the given number to mixed number with integer part 1 and a decimal fractional part. Then,

$$0.485/0.25 = 1.94 = 1 + 0.5 + 0.25 + 0.125 + 0.0625 + 0.03125 + \ldots$$
$$= 1 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + \ldots = (1.11111\ldots)_2$$

Since we divided the given number by $0.25 = 2^{-2}$, we must have

$$2^{(E-127)} = 2^{-2}$$

from which

$$E - 127 = -2$$

or

$$E = (125)_{10} = (01111101)_2$$

Thus, the number $(0.485)_{10}$ in IEEE single precision floating point representation is

$$\underbrace{0}_{S} \qquad \underbrace{01111101}_{E} \qquad \underbrace{111111\ldots\ldots\ldots\ldots}_{F}$$

$$S = + \qquad E = (125)_{10} \qquad F = (1.11111\ldots)_2$$

Check:

$$(\text{Value})_{10} = (-1)^0 \cdot 2^{(125-127)} \cdot (1.11111\ldots)_2$$

$$= 1 \times 2^{-2} \times (1 + 0.5 + 0.25 + 0.125 + 0.0625 + 0.03125 + \ldots) = 0.485$$

6.

The results are the same as in Exercise 5 except that the sign bit is one, and thus the number $(-0.485)_{10}$ in IEEE single precision floating point representation is

$$\underbrace{1}_{S} \qquad \underbrace{01111101}_{E} \qquad \underbrace{111111\ldots\ldots\ldots\ldots}_{F}$$

$$S = - \qquad E = (125)_{10} \qquad F = (1.11111\ldots)_2$$

Chapter 4

Binary Codes

This chapter describes the most commonly used binary codes. In particular, we will discuss the Binary Coded Decimal (BCD), the Excess-3 Code, the 2*421 Code, the Gray Code, and the American Standard Code for Information Interchange (ASCII) code. We will also discuss parity bits.

4.1 Encoding

In general, *encoding* is the process of putting a message into a code. In this text, encoding will refer to a process where the decimal numbering system as well as other *alphanumerics* (combinations of numbers and letters) are represented by the digits of the binary numbering system, that is 0 and 1. In this section we will introduce four different codes where 4 bits are used to represent a single decimal character.

4.1.1 Binary Coded Decimal (BCD)

The *Binary Coded Decimal* (BCD) uses 4 bits to represent the decimal numbers 0 through 9 and these are shown in the Table 4.1. This code is also known as 8421 code because the digits 8, 4, 2, and 1 represent the weight of its bits position. In other words, the BCD is a *weighted code*.

TABLE 4.1 The BCD code

Decimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

We observe that only ten of the sixteen (2^4) possible combinations are used in BCD; the remaining six combinations 1010, 1011, 1100, 1101, 1110, and 1111 are invalid in BCD. The designa-

tions A through F to represent these numbers are valid only in the hexadecimal number system. Since in BCD we require four bits to represent a single decimal character, a decimal number between 10 and 99 must be represented with 8 bits, a decimal number between 100 and 9999 will require 12 bits in BCD, and so on. In general, a decimal number with n digits in BCD must be represented by $4n$ bits.

Note: In our subsequent designations spaces will be added to indicate the BCD groupings.

Example 4.1

Express $(10)_{10}$, $(279)_{10}$, and $(53904)_{10}$ in BCD.

Solution:

$$(10)_{10} = (0001\ 0000)_{BCD}$$

$$(279)_{10} = (0010\ 0111\ 1001)_{BCD}$$

$$(53904)_{10} = (0101\ 0011\ 1001\ 0000\ 0100)_{BCD}$$

4.1.2 The Excess-3 Code

The *Excess-3 code* also uses 4 bits to represent the decimal numbers 0 through 9 and these are shown in the Table 4.2.

TABLE 4.2 The Excess-3 code

Decimal	Excess-3
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

The Excess-3 Code derives its name from the fact that each decimal representation in Excess-3 code is larger than the BCD code by three. The advantage of the Excess-3 code over the BCD code is that the Excess-3 code is a *self-complementing code* as illustrated below.

From Table 4.2 we observe that

$$(1)_{10} = (0100)_{Excess-3}$$

and

$$(8)_{10} = (1011)_{Excess-3}$$

These relations reveal that the left sides are the nines-complement of each other whereas the right sides are the ones-complement of each other. Also,

$$(4)_{10} = (0111)_{Excess-3}$$

and

$$(5)_{10} = (1000)_{Excess-3}$$

Again, we observe that the left sides are the nines-complement of each other whereas the right sides are the ones-complement of each other. Therefore, the Excess-3 code can be used to perform subtraction of decimal numbers using the nines-complement method. However, the Excess-3 code is not a weighted code as BCD. For instance,

$$\left(\underset{4}{0111} + \underset{2}{0101} \neq \underset{7}{1010}\right)_{Excess-3}$$

whereas

$$\left(\underset{4}{0100} + \underset{5}{0101} = \underset{9}{1001}\right)_{BCD}$$

4.1.3 The 2*421 Code

The *2*421 code* also uses 4 bits to represent the decimal numbers 0 through 9 and these are shown in the Table 4.3.

TABLE 4.3 The 2*421 code

Decimal	2*421
0	0000
1	0001
2	0010
3	0011
4	0100
5	1011
6	1100
7	1101
8	1110
9	1111

The 2*421 code incorporates the advantages of both the BCD and the Excess-3 code, that is, it is

both self-complementing and weighted code, for example,

$$(2)_{10} = (0010)_{2*421}$$

$$(7)_{10} = (1101)_{2*421}$$

We observe that the left sides are the nines-complements of each other and the right sides are the ones-complements of each other. Moreover,

$$\left(\underset{4}{0100} + \underset{5}{1011} = \underset{9}{1111} \right)_{2*421}$$

4.1.4 The Gray Code

The *Gray code* also uses 4 bits to represent the decimal numbers 0 through 9 and these are shown in the Table 4.4.

TABLE 4.4 The Gray code

Decimal	Gray
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101

The Gray Code is neither a self-complementing code nor a weighted code. We observe that in this code only a single bit changes between successive numbers. This is very desirable in certain applications such as optical or mechanical shaft position encoders, and digital-to-analog conversion. To generate this code we start with all zeros and subsequently change the lsb that will produce a new value.

For convenience, we have grouped all 4 codes into Table 4.5

*TABLE 4.5 The BCD, Excess-3, 2*421, and Gray codes*

Decimal	BCD	Excess-3	2*421	Gray
0	0000	0011	0000	0000
1	0001	0100	0001	0001
2	0010	0101	0010	0011
3	0011	0110	0011	0010
4	0100	0111	0100	0110
5	0101	1000	1011	0111
6	0110	1001	1100	0101
7	0111	1010	1101	0100
8	1000	1011	1110	1100
9	1001	1100	1111	1101

4.2 The American Standard Code for Information Interchange (ASCII) Code

Digital computers use alphanumeric codes, that is, codes consisting of both alphabetic and numeric characters. The *American Standard Code for Information Interchange* (*ASCII*) code is the most widely accepted. It was proposed by the American National Standards Institute (ANSI).

The table of the next page is referred to as *seven-bit ASCII* and represents 128 (2^7) characters assigned to numbers, letters, punctuation marks, and the most special characters. The standard 7-bit character representation with b_7 the high-order bit and b_1 the low-order bit is shown in Figure 4.1.

As an example, the bit representation of the letter *j* is

b_7	b_6	b_5	b_4	b_3	b_2	b_1
1	1	0	1	0	1	0

A description of the first 32 ASCII characters, often referred to as *control codes*, follows.

NUL – A character code with a null value; literally, a character meaning "nothing." Although it is real in the sense of being recognizable, occupying space internally in the computer, and being sent or received as a character, a NUL character displays nothing, takes no space on the screen or on paper, and causes no specific action when sent to a printer. In ASCII, NUL is represented by the character code 0. It can be addressed like a physical output device (such as a printer) but it discards any information sent to it.

SOH – Start of Heading

STX – Start of Text

BIT NUMBERS								0_0	0_0	0_1	0_1	1_0	1_0	1_1	1_1
b_7	b_6	b_5	b_4	b_3	b_2	b_1	COLUMN / ROW	0	1	2	3	4	5	6	7
			0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
			0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
			0	0	1	0	2	STX	DC2	"	2	B	R	b	r
			0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
			0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
			0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
			0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
			0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
			1	0	0	0	8	BS	CAN	(	8	H	X	h	x
			1	0	0	1	9	HT	EM	)	9	I	Y	i	y
			1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
			1	0	1	1	11	VT	ESC	+	;	K	[	k	{
			1	1	0	0	12	FF	FS	,	<	L	\	l	|
			1	1	0	1	13	CR	GS	-	=	M	]	m	}
			1	1	1	0	14	SO	RS	.	>	N	^	n	~
			1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Figure 4.1. The Standard ASCII Code

ETX – Marks the end of a text file. End-of-text does not necessarily mean end of transmission; other information, such as error-checking or transmission-control characters, can be included at the end of the file. In ASCII, end-of-text is represented by the decimal value 3 (hexadecimal 03).

EOT – End of Transmission (Not the same as ETB)

ENQ – Enquiry - A control code transmitted from one station to request a response from the receiving station. In ASCII, the enquiry character is designated by decimal value 5 (hexadecimal 05).

ACK – Acknowledgement – A message sent by the receiving unit to the sending station or computer indicating either that the unit is ready to receive transmission or that a transmission was received without error.

BEL – Bell – Causes teletype machines to ring a bell. Causes a beep in many common terminals and terminal emulation programs.

BS – Backspace – Moves the cursor (or print head) move backwards (left) one space.

TAB – Horizontal Tab – Moves the cursor (or print head) right to the next tab stop. The spacing of tab stops is dependent on the device, but is often either 8 or 10.

LF – Linefeed – Tells a computer or printer to advance one line below the current line without moving the position of the cursor or print head.

VT – Vertical Tab

FF – Form Feed – Advances paper to the top of the next page (if the output device is a printer).

CR – Carriage Return – Tells a printer to return to the beginning of the current line. It is similar to the return on a typewriter but does not automatically advance to the beginning of a new line. For example, a carriage-return character alone, received at the end of the words "This is a sample line of text" would cause the cursor or printer to return to the first letter of the word "This." In the ASCII character set, the CR character has the decimal value of 13 (hexadecimal 0D).

SO – Shift Out – Switches output device to alternate character set.

SI – Shift In – Switches output device back to default character set.

DLE – Data Link Escape

DC1 – Device Control 1

DC2 – Device Control 2

DC3 – Device Control 3

DC4 – Device Control 4

NAK – Negative Acknowledgement – A control code, ASCII character 21 (hexadecimal 15H), transmitted to a sending station or computer by the receiving unit as a signal that transmitted information has arrived incorrectly.

SYN – Synchronous – A character used in synchronous (timed) communications that enables the sending and receiving devices to maintain the same timing.

ETB – End of Transmission Block) – Not the same as EOT

CAN – Cancel

EM – End of Medium

SUB – Substitute

ESC – Escape – Usually indicates the beginning of an escape sequence (a string of characters that give instructions to a device such as a printer). It is represented internally as character code 27 (hexadecimal 1B).

FS – File Separator

GS – Group Separator

RS – Record Separator

US – Unit Separator

DEL – Delete

4.3 The Extended Binary Coded Decimal Interchange Code (EBCDIC)

The *Extended Binary Coded Decimal Interchange Code* (EBCDIC) also known as *Extended ASCII Character Set* consists of 128 additional decimal numbers, that is, 256 decimal numbers. The range from 128 through 255 represents additional special, mathematical, graphic, and foreign characters. EBCDIC was developed by IBM for use with IBM mainframes.

4.4 Parity Bits

A *parity bit* is an additional bit that can be added to a group of zeros and ones to make the parity of the group odd or even. An odd parity exists when the total number of ones including the parity bit is an odd number such as 1, 3, 5, and so on, and an even parity exists when the total number of ones including the parity bit is an even number such as 2, 4, 6, and so on. The parity bit can be placed either in front of the msb or after the lsb.

Example 4.2

Express the sentence SO LONG. in ASCII code without parity.

Solution:

$$(\text{SO LONG.})_{\text{English}} =$$

$$\left(\underset{\text{S}}{1010011}\ \underset{\text{O}}{1001111}\ \underset{\text{space}}{0100000}\ \underset{\text{L}}{1001100}\ \underset{\text{O}}{1001111}\ \underset{\text{N}}{1001110}\ \underset{\text{G}}{1000111}\ \underset{\text{period}}{0101100}\right)_{\text{ASCII}}$$

Example 4.3

Express the word Print in ASCII code with even parity and place the parity bit at the msb position.

Solution:

$$(\text{Print})_{\text{English}} =$$

$$\left(\underset{\text{P}}{01010000}\ \underset{\text{r}}{01110010}\ \underset{\text{i}}{01101001}\ \underset{\text{n}}{11101110}\ \underset{\text{t}}{01110100}\right)_{\text{ASCII}}$$

The parity bits are very useful in the data transmission operation. Thus, when a message is sent to a particular location, the receiving end checks to see that each binary word (representing) a particular alphanumeric character contains either an even or an odd number of ones. Accordingly, if even parity is used and the number of ones received is odd, the receiving end rejects that binary word since it is known that an error has occurred. Of course, if two bit errors occur in a single binary word, the parity will appear to be correct and the receiving end will decode that word incorrectly. However, the probability of two errors occurring in a single binary word is very remote. The parity bit is discarded when the message is received.

4.5 Error Detecting and Correcting Codes

Error detecting and correcting codes are topics discussed in Information Theory textbooks. Here, we will present an example of a simple systematic code in which a single check digit is added as a means of detecting an odd number of errors in a coded word. The information digits are arranged in a two dimensional array as shown below where an even-parity-check digit has been added to each row and each column. In addition, checks are also carried out on check digits.

	Information Bits						
Information Bits	1	0	1	1	0	1	0
	0	1	1	1	0	1	0
	1	0	1	0	0	1	1
	1	0	0	0	1	1	1
	0	0	1	1	0	1	1
	1	0	1	1	1	0	0
Row Checks	0	1	1	0	0	1	1
		Column Checks					

As shown above, parity bits for even parity have been added on the last row and the right column. Now, suppose that the 0 on the fourth row and fourth column was erroneously transmitted as 1. This error will be detected at the receiving station since the row and column checks will both indicate odd instead of even parity. Since the error appears to be on the fourth row and fourth column, their intersection will point to the erroneous bit of 1, and a correction will be made by replacing it with a 0.

4.6 Cyclic Codes

Cyclic codes are a class of codes in which a vector W defined as $W = W_1, W_2, \ldots, W_n$ when cyclically shifted one unit to the right, produces a vector W' defined as $W' = W_n, W_1, \ldots, W_{n-1}$ which is also a word of the code. These codes are very practical because of the simplicity with which can be synthesized and can be encoded and decoded using simple feedback shift registers.

A class of cyclic codes is the well-known *Bose-Chauduri codes* and these codes are very effective in detecting and correcting randomly occurring multiple errors. The Bose-Chauduri codes is a lengthy topic and will not be discussed here.

Errors occur in telecommunications systems primarily from disturbances (noise) that are relatively long in time duration. Thus, a noise burst of 0.01 second is not uncommon, and if it occurs during a 1,200 bit per second data transmission, we would expect (at worst case) to receive 12 erroneous data bits known as an error burst of 12 bits long. A *cyclic redundancy check* code known as *CRC-12* has a 99.955% chance of detecting error bursts of 12 bits in length.

To check for errors, a cyclic redundancy check or simply CRC, uses a math calculation to generate a number based on the data transmitted. The sending device performs the calculation before transmission and sends its result to the receiving device. The receiving device repeats the same calculation after transmission. If both devices obtain the same result, it is assumed that the transmission was error-free. The procedure is known as a redundancy check because each transmission includes not only data but extra (redundant) error-checking values.

CRCs add several bits (up to about 25) to the message to be transmitted. The total data block (message plus added bits) to be transmitted, is considered as one lengthy binary polynomial* denoted as $T(x)$. At the *transmitting end* the message block (the actual message without the added bits) expressed as a polynomial in powers of x. It is referred to as the *message polynomial* and it is denoted as $M(x)$. The message polynomial $M(x)$ is divided by a fixed polynomial referred to as the *generator polynomial* and denoted as $G(x)$ that is known to both the transmitter and the receiver. This division produces a *quotient polynomial* denoted as $Q(x)$ and a *remainder polynomial* denoted as $R(x)$.

The remainder polynomial $R(x)$ also known as the *checksum*, is added to the message polynomial to form the encoded transmitted polynomial $T(x)$

At the *receiving end* the received data block is divided by the same generator polynomial $G(x)$. If it is found that there is no remainder. i.e., remainder is zero, the message was transmitted correctly. However, if the remainder is non-zero, an error has occurred. In this case either retransmission is requested, or error correction needs to be performed. The non-zero polynomial at the receiving end is referred to as the *error polynomial* and it is denoted as $E(x)$.

* *The serial data bits of the various code words in CRCs are usually represented as polynomials in the form* $x^n + x^{n-1} + \ldots + x^2 + x + 1$ *where* x^n *represents the most significant bit (msb) and* x^0 *the least significant bit (lsd). Missing terms represent zeros; for instance the binary word 10101 is represented by the polynomial* $x^4 + x^2 + 1$. *We observe that the degree of the polynomial is one less than the number of bits in the code word.*

The addition or subtraction operation when calculating polynomials in CRCs is the *modulo-2 addition* and it is performed by Exclusive-OR gates discussed in Chapter 6. This addition is denined by the table below.

$$0 \oplus 0 = 0$$
$$0 \oplus 1 = 1$$
$$1 \oplus 0 = 1$$
$$1 \oplus 1 = 0$$

Modulo-2 subtraction is identical to modulo-2 addition.

For convenience, let us review the definitions of the polynomials mentioned above.

M(x) = message polynomial consisting of m bits to be transmitted
G(x) = generator polynomial of fixed length known by transmitter and receiver
T(x) = polynomial consisting of t bits to be transmitted
Q(x) = quotient polynomial
R(x) = remainder polynomial
E(x) = error polynomial

The transmitted polynomial $T(x)$ is obtained as follows:

1. The message polynomial $M(x)$ is shifted c bits to the left where c is the highest degree of the generator polynomial $G(x)$. The shifted data block is thus expressed as $M(x)x^c$.

2. The new polynomial $M(x)x^c$ is now divided by the generator polynomial $G(x)$, and the remainder (checksum) of this division is added to $M(x)x^c$ to form the transmitted polynomial $T(x)$. The transmitted polynomial $T(x)$ obtained by this procedure can always be divided by the polynomial $G(x)$ as proved below.

Proof:

$$\frac{M(x)x^c}{G(x)} = Q(x) + \frac{R(x)}{G(x)} \tag{4.1}$$

Multiplication of (4.1) by $G(x)$ yields

$$M(x)x^c = Q(x)G(x) + R(x) \tag{4.2}$$

or

$$M(x)x^c - R(x) = Q(x)G(x) \tag{4.3}$$

Since subtraction is identical to addition, (4.3) can be expressed as

$$T(x) = M(x)x^c + R(x) = Q(x)G(x) \tag{4.4}$$

or

$$\frac{T(x)}{G(x)} = Q(x) + \text{Zero Remainder} \tag{4.5}$$

Example 4.4

Let the message and generator polynomials be

$$M(x) = 101101 \rightarrow x^5 + x^3 + x^2 + 1$$

$$G(x) = 11001 \rightarrow x^4 + x^3 + 1$$

Since the highest degree of $G(x)$ is 4, we shift $M(x)$ four bits to the left. In polynomial form, this is done by forming the product $M(x)x^c$. For this example,

$$M(x)x^4 = (x^5 + x^3 + x^2 + 1)(x^4 + x^3 + 1) = x^9 + x^7 + x^6 + x^4 \rightarrow 1011010000$$

Next, we divide this product by $G(x)$

Divisor $x^4 + x^3 + 1$ | Quotient $x^5 + x^4 + x^2$ | Dividend $x^9 + x^7 + x^6 + x^4$

$x^9 + x^8 + x^5$

$x^8 + x^7 + x^6 + x^5 + x^4$

$x^8 + x^7 + x^4$

$x^6 + x^5$

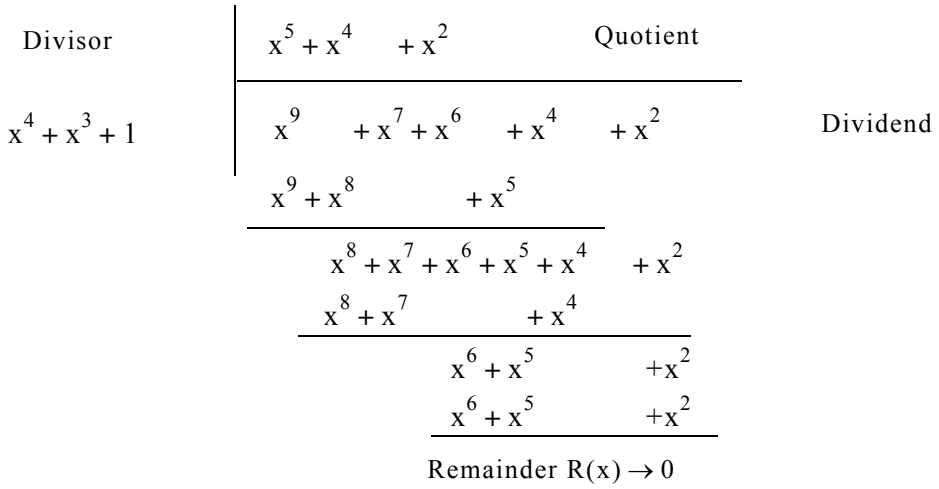
$x^6 + x^5 + x^2$

Remainder $R(x) \rightarrow x^2$

Now, we add $R(x)$ to $M(x)x^4$ to get the transmitted polynomial

$$T(x) = M(x)x^4 + R(x) = x^9 + x^7 + x^6 + x^4 + x^2 \rightarrow 1011010100$$

If there is no error at the receiver, this message will produce no remainder when divided by $G(x)$. A check procedure is shown below.



4.7 Summary

- Encoding is the process of putting a message into a code.
- Alphanumerics are combinations of numbers and letters represented by the digits of the binary numbering system.
- The Binary Coded Decimal (BCD) uses 4 bits to represent the decimal numbers 0 through 9. This code is also known as 8421 code. The BCD is a weighted code. Only ten of the sixteen (2^4) possible combinations are used in BCD; the remaining six combinations 1010, 1011, 1100, 1101, 1110, and 1111 are invalid in BCD. A decimal number between 10 and 99 must be represented with 8 bits in BCD, a decimal number between 100 and 9999 will require 12 bits in BCD, and so on. In general, a decimal number with n digits in BCD must be represented by $4n$ bits.
- The Excess-3 code also uses 4 bits to represent the decimal numbers 0 through 9. This code derives its name from the fact that each decimal representation in Excess-3 code is larger than the BCD code by three. The advantage of the Excess-3 code over the BCD code is that the Excess-3 code is a self-complementing code. The Excess-3 code is not a weighted code.
- Thee 2*421 code is another code that uses 4 bits to represent the decimal numbers 0 through 9. This code incorporates the advantages of both the BCD and the Excess-3 code, that is, it is both self-complementing and weighted code.
- The Gray code also uses 4 bits to represent the decimal numbers 0 through 9. The Gray Code is neither a self-complementing code nor a weighted code. In this code only a single bit changes between successive numbers. This is very desirable in certain applications such as optical or mechanical shaft position encoders, and digital-to-analog conversion. To generate this code we start with all zeros and subsequently change the lsb that will produce a new value.
- The American Standard Code for Information Interchange (ASCII) code is the most widely accepted code for representing alphanumeric characters. It was proposed by the American National Standards Institute (ANSI). The seven-bit ASCII represents 128 (2^7) characters assigned to numbers, letters, punctuation marks, and the most special characters.
- The Extended Binary Coded Decimal Interchange Code (EBCDIC) also known as Extended ASCII Character Set consists of 128 additional decimal numbers, that is, 256 decimal numbers. The range from 128 through 255 represents additional special, mathematical, graphic, and foreign characters. EBCDIC was developed by IBM for use with IBM mainframes.
- A parity bit is an additional bit that can be added to a group of zeros and ones to make the parity of the group odd or even. An odd parity exists when the total number of ones including the parity bit is an odd number such as 1, 3, 5, and so on, and an even parity exists when the total

number of ones including the parity bit is an even number such as 2, 4, 6, and so on. The parity bit can be placed either in front of the msb or after the lsb. The parity bits are very useful in the data transmission operation.

- Error detecting and correcting codes are codes that can detect and also correct errors occurring during the transmission of binary data.
- Cyclic codes are very practical because of the simplicity with which can be synthesized and can be encoded and decoded using simple feedback shift registers. A class of cyclic codes is the well-known Bose-Chauduri codes and these codes are very effective in detecting and correcting randomly occurring multiple errors.
- A cyclic redundancy check or simply CRC, adds several bits to the message to be transmitted. The total data block (message plus added bits) to be transmitted, is considered as one lengthy binary polynomial denoted as $T(x)$. At the transmitting end the message block (the actual message without the added bits) expressed as a polynomial in powers of x. It is referred to as the message polynomial and it is denoted as $M(x)$. The message polynomial $M(x)$ is divided by a fixed polynomial referred to as the generator polynomial and denoted as $G(x)$ that is known to both the transmitter and the receiver. This division produces a quotient polynomial denoted as $Q(x)$ and a remainder polynomial denoted as $R(x)$. The remainder polynomial $R(x)$ also known as the checksum, is added to the message polynomial to form the encoded transmitted polynomial $T(x)$. At the receiving end the received data block is divided by the same generator polynomial $G(x)$. If it is found that there is no remainder. i.e., remainder is zero, the message was transmitted correctly. However, if the remainder is non-zero, an error has occurred. In this case either retransmission is requested, or error correction needs to be performed. The non-zero polynomial at the receiving end is referred to as the error polynomial and it is denoted as $E(x)$.
- The addition or subtraction operation when calculating polynomials in CRCs is the modulo-2 addition and it is performed by Exclusive-OR gates.

4.8 Exercises

1. Express $(49)_{10}$, $(827)_{10}$, and $(27852)_{10}$ in BCD.

2. Express $(674.027)_{10}$ in BCD.

3. Express $(674.027)_{10}$ in Excess-3 code.

4. Express $(809.536)_{10}$ in 2*421 code.

5. Express $(809.536)_{10}$ in Gray code.

6. Gray-to-binary conversion can be performed with the following procedure:

1. First (msb) bit of binary is same as the msb of Gray.
2. Second bit of binary is obtained by modulo-2 addition of the second bit of Gray with the first bit of binary obtained in Step 1 above.
3. Third bit of binary is obtained by modulo-2 addition of the third bit of Gray with the second bit of binary obtained in Step 2 above, and so on.

Using the above procedure, convert the Gray code word 1110 1011 into its binary equivalent.

7. Express the sentence so long. in ASCII code with odd parity, and place the parity bit at the lsd position.

4.9 Solutions to End-of-Chapter Exercises

1.

$$(49)_{10} = (0100\ 1001)_{BCD}$$

$$(827)_{10} = (1000\ 0010\ 0111)_{BCD}$$

$$(27852)_{10} = (0010\ 0111\ 1000\ 0101\ 0010)_{BCD}$$

2.

$$(674.027)_{10} = (0110\ 0111\ 0100\ .\ 0000\ 0010\ 0111)_{BCD}$$

3.

$$(674.027)_{10} = (1001\ 1010\ 0111\ .\ 0011\ 0101\ 1010)_{EXCESS-3}$$

4.

$$(809.536)_{10} = (1110\ 0000\ 1111\ .\ 1011\ 0011\ 1100)_{2*421}$$

5.

$$(809.536)_{10} = (1100\ 0000\ 1101\ .\ 0111\ 0010\ 0101)_{GRAY}$$

6.

$$(1110\ 1011)_{GRAY} = (10110010)_{BINARY}$$

7.

$$(\text{so long.})_{English} =$$

$$\left(\underset{s}{\underline{11100110}}\ \underset{o}{\underline{11011111}}\ \underset{space}{\underline{01000000}}\ \underset{l}{\underline{11011001}}\ \underset{o}{\underline{11011111}}\ \underset{n}{\underline{11011100}}\ \underset{g}{\underline{11001110}}\ \underset{period}{\underline{01011000}}\right)_{ASCII}$$

Chapter 5

Fundamentals of Boolean Algebra

This chapter begins with the basic logic operations and continues with the fundamentals of Boolean algebra and the basic postulates and theorems as applied to electronic logic circuits. Truth tables are defined and examples are given to illustrate how they can be used to prove Boolean algebra theorems or equivalent expressions.

5.1 Basic Logic Operations

The following three logic operations constitute the three basic logic operations performed by a digital computer.

1. *Conjunction* (or *logical product*) commonly referred to as the *AND operation* and denoted with the dot (.) symbol between variables.
2. *Disjunction* (or *logical sum*) commonly called the *OR operation* and denoted with the plus (+) symbol between variables
3. *Negation* (or *complementation* or *inversion*) commonly called *NOT operation* and denoted with the bar (–) symbol above the variable.

5.2 Fundamentals of Boolean Algebra

This section introduces the basics of Boolean algebra. We need to know these to understand Chapter 6 and all subsequent chapters in this text.

5.2.1 Postulates

Postulates (or *axioms*) are propositions taken as facts; no proof is required. A well-known axiom states that the shortest distance between two points is a straight line.

1. Let X be a variable. Then, $X = 0$ or $X = 1$. If $X = 0$, then $\overline{X} = 1$, and vice-versa.
2. $0 \cdot 0 = 0$
3. $0 \cdot 1 = 1 \cdot 0 = 0$
4. $1 \cdot 1 = 1$
5. $0 + 0 = 0$
6. $0 + 1 = 1 + 0 = 1$
7. $1 + 1 = 1$

5.2.2 Theorems

1. *Commutative laws*
   a. $A \cdot B = B \cdot A$
   b. $A + B = B + A$
2. *Associative laws*
   a. $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
   b. $(A + B) + C = A + (B + C)$
3. *Distributive laws*
   a. $A \cdot (B + C) = A \cdot B + A \cdot C$
   b. $A + (B \cdot C) = (A + B) \cdot (A + C)$
4. *Identity laws*
   a. $A \cdot A = A$
   b. $A + A = A$
5. *Negation laws*
   a. $\overline{(A)} = \overline{A}$
   b. $\overline{(\overline{A})} = \overline{\overline{A}} = A$
6. *Redundancy laws*
   a. $A \cdot (A + B) = A$
   b. $A + (A \cdot B) = A$
7.
   a. $0 \cdot A = 0$
   b. $1 \cdot A = A$
   c. $0 + A = A$
   d. $1 + A = 1$
8.
   a. $A \cdot \overline{A} = 0$
   b. $A + \overline{A} = 1$

9.

a. $A \cdot (\overline{A} + B) = A \cdot B$

b. $A + (\overline{A} \cdot B) = A + B$

10. DeMorgan's theorems

a. $\overline{A \cdot B} = \overline{A} + \overline{B}$

b. $\overline{A + B} = \overline{A} \cdot \overline{B}$

5.3 Truth Tables

A *truth table* provides a listing of every possible combination of inputs and lists the output of a logic circuit which results from each input combination. A truth table can be used to prove that one logical expression is equal to another or to prove theorems. When a proof is accomplished by the use of the truth table it is called *proof by perfect induction*.

Example 5.1

Use proof by perfect induction to show the validity of Theorem 6.a, i.e., $A + (A \cdot B) = A$.

Proof:

A	B	$A \cdot B$	$A + A \cdot B$		
0	0	0	0	←	Using Postulates 2 and 5
0	1	0	0	←	Using Postulates 3 and 5
1	0	0	1	←	Using Postulates 3 and 6
1	1	1	1	←	Using Postulates 4 and 7
↑			↑	←	The first and fourth columns are identical. Thus, $A + (A \cdot B) = A$

Note: In our subsequent discussions an uncomplemented variable will represent a logical 1 and a complemented variable will represent a logical 0. Thus, if $A = 1$, then $\overline{A} = 0$. Conversely, if $\overline{B} = 1$, then $B = 0$.

Example 5.2

Use proof by perfect induction to show the validity of Theorem 9.b, i.e., $A + (\overline{A} \cdot B) = A + B$.

Proof:

A	B	$\overline{A} \cdot B$	$A + (\overline{A} \cdot B)$	$A + B$		
0	0	0	0	0	←	Using Postulates 2 and 5
0	1	1	1	1	←	Using Postulates 4 and 7
1	0	0	1	1	←	Using Postulates 2 and 6
1	1	0	1	1	←	Using Postulates 3 and 6
			↑	↑	←	The fourth and fifth columns are identical. Thus, $A + (\overline{A} \cdot B) = A + B$

Some Boolean expressions can by simplified by using the postulates and theorems of Boolean algebra without truth tables.

Example 5.3

Simplify the Boolean expression $C = \overline{A} \cdot B + A \cdot B + \overline{A} \cdot \overline{B}$

Solution:

From Theorem 3.a, $A \cdot B + A \cdot C = A \cdot (B + C)$ and for this example, $\overline{A} \cdot B + A \cdot B = (\overline{A} + A)B$ and from Theorem 8.b, $A + \overline{A} = 1$. Then, the given expression reduces to

$$C = 1 \cdot B + \overline{A} \cdot \overline{B}$$

Next, from Theorem 7.b, $1 \cdot B = B$ and thus

$$C = B + \overline{A} \cdot \overline{B}$$

Finally, from Theorem 9.b, $A + (\overline{A} \cdot B) = A + B$ or $B + (\overline{B} \cdot A) = A + B$. Therefore,

$$C = \overline{A} + B$$

5.4 Summary

- The following three logic operations constitute the three basic logic operations performed by a digital computer.
  1. Conjunction (or logical product) commonly referred to as the AND operation and denoted with the dot (.) symbol between variables.
  2. Disjunction (or logical sum) commonly called the OR operation and denoted with the plus (+) symbol between variables
  3. Negation (or complementation or inversion) commonly called NOT operation and denoted with the bar (–) symbol above the variable.
- The following postulates and theorems will be used throughout this text:

A. Postulates

1. Let X be a variable. Then, $X = 0$ or $X = 1$. If $X = 0$, then $\overline{X} = 1$, and vice-versa.
2. $0 \cdot 0 = 0$
3. $0 \cdot 1 = 1 \cdot 0 = 0$
4. $1 \cdot 1 = 1$
5. $0 + 0 = 0$
6. $0 + 1 = 1 + 0 = 1$
7. $1 + 1 = 1$

B. Theorems

1. Commutative laws
   a. $A \cdot B = B \cdot A$
   b. $A + B = B + A$
2. Associative laws
   a. $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
   b. $(A + B) + C = A + (B + C)$
3. *Distributive laws*
   a. $A \cdot (B + C) = A \cdot B + A \cdot C$
   b. $A + (B \cdot C) = (A + B) \cdot (A + C)$

4. Identity laws
   a. $A \cdot A = A$
   b. $A + A = A$
5. Negation laws
   a. $\overline{(A)} = \overline{A}$
   b. $\overline{(\overline{A})} = \overline{\overline{A}} = A$
6. Redundancy laws
   a. $A \cdot (A + B) = A$
   b. $A + (A \cdot B) = A$
7.
   a. $0 \cdot A = 0$
   b. $1 \cdot A = A$
   c. $0 + A = A$
   d. $1 + A = 1$
8.
   a. $A \cdot \overline{A} = 0$
   b. $A + \overline{A} = 1$
9.
   a. $A \cdot (\overline{A} + B) = A \cdot B$
   b. $A + (\overline{A} \cdot B) = A + B$
10. DeMorgan's theorems
    a. $\overline{A \cdot B} = \overline{A} + \overline{B}$
    b. $\overline{A + B} = \overline{A} \cdot \overline{B}$

- A truth table provides a listing of every possible combination of inputs and lists the output of a logic circuit which results from each input combination.
- When a proof is accomplished by the use of the truth table it is called proof by perfect induction.

5.5 Exercises

1. Use proof by perfect induction to show the validity of DeMorgan's theorems.

2. Simplify the Boolean expression

$$C = A\overline{B} + \overline{A} + B$$

3. Use proof by perfect induction to verify the result of Exercise 2.

5.6 Solutions to End-of-Chapter Exercises

1.

a.

A	B	$\overline{A \cdot B}$	$\overline{A} + \overline{B}$		
0	0	1	1		
0	1	0	0		
1	0	0	0		
1	1	0	0		
		↑	↑	←	The third and fourth columns are identical. Thus, $\overline{A \cdot B} = \overline{A} + \overline{B}$

b.

A	B	$\overline{A + B}$	$\overline{A} \cdot \overline{B}$		
0	0	1	1		
0	1	0	0		
1	0	0	0		
1	1	0	0		
		↑	↑	←	The third and fourth columns are identical. Thus, $\overline{A + B} = \overline{A} \cdot \overline{B}$

2.

From Theorem 9(a)

$$B + A\overline{B} = A + B$$

Thus,

$$C = A + B + \overline{A}$$

From Theorem 8(a)

$$A + \overline{A} = 1$$

Then,

$$C = 1 + B$$

and from Theorem 7(c)

$$C = 1$$

3.

A	B	$A \cdot \overline{B}$	$\overline{A} + B$	$C = A \cdot \overline{B} + \overline{A} + B$
0	0	0	1	1
0	1	0	1	1
1	0	1	0	1
1	1	0	1	1
				↑

← The fifth column shows that C is 1 for all $A \cdot \overline{B} + \overline{A} + B$

Chapter 6

Minterms and Maxterms

This chapter introduces two standard forms of expressing Boolean functions, the minterms and maxterms, also known as standard products and standard sums respectively. A procedure is also presented to show how one can convert one form to the other.

6.1 Minterms

In Chapter 5 we learned that a binary variable may appear either as A or $\overline{A}$. Also, if two binary variables A and B are ANDed, the four $(2^2 = 4)$ possible combinations are $\overline{A}\overline{B}$,* $\overline{A}B$, $A\overline{B}$, and AB, and each of these defines a distinct area in a *Vern diagram*† as shown in Figure 6.1.

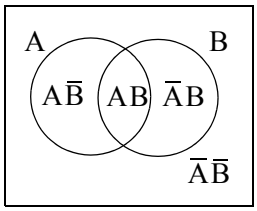


Figure 6.1. Vern Diagram for the ANDing of the variables A and B

In Figure 6.1 the left circle is denoted as A, the right circle as B, the common area as AB, the area of circle A which excludes B as $A\overline{B}$, the area of circle B which excludes A as $\overline{A}B$, and the area outside the two circles as $\overline{A}\overline{B}$. These four terms are referred to as *minterms* or *standard products* where the word product here means the ANDing of these variables. In other words, a minterm is composed of two or more variables or their complements ANDed together.

In general, for n variables there are 2^n minterms designated as m_j where the subscript j represents the decimal equivalent of the binary number of the minterm. Table 6.1 shows the 8 minterms for the variables A, B, and C, and their designations.

* *Henceforth, for simplicity the dot between two variables ANDed together will be omitted.*

† *Vern diagrams are discussed in detail in Boolean algebra textbooks. They will not be discussed in this text.*

TABLE 6.1 Minterms for 3 variables and their designation

A	B	C	Minterm	Designation
0	0	0	$\overline{A}\overline{B}\overline{C}$	m_0
0	0	1	$\overline{A}\overline{B}C$	m_1
0	1	0	$\overline{A}B\overline{C}$	m_2
0	1	1	$\overline{A}BC$	m_3
1	0	0	$A\overline{B}\overline{C}$	m_4
1	0	1	$A\overline{B}C$	m_5
1	1	0	$AB\overline{C}$	m_6
1	1	1	ABC	m_7

As shown in Table 6.1, each minterm is obtained from an ANDed term of n variables with each variable being complemented if the corresponding bit is a logical 0, and uncomplemented if the corresponding bit is logical 1.

6.2 Maxterms

In analogy with minterms, n variables forming an ORed term, with each variable complemented or uncomplemented, form 2^n possible combinations referred to as *maxterms* or *standard products* where the word sum here means the ORing of these variables. In other words, a maxterm is composed of two or more variables or their complements ORed together.

In general, for n variables there are 2^n maxterms designated as M_j where the subscript j represents the decimal equivalent of the binary number of the maxterm. Table 6.2 shows the 8 maxterms for the variables A, B, and C, and their designations.

As shown in Table 6.2, each maxterm is obtained from an ORed term of n variables with each variable being uncomplemented if the corresponding bit is a logical 0, and complemented if the corresponding bit is logical 1.

TABLE 6.2 Maxterms for 3 variables and their designation

A	B	C	Maxterm	Designation
0	0	0	$A + B + C$	M_0
0	0	1	$A + B + \overline{C}$	M_1
0	1	0	$A + \overline{B} + C$	M_2
0	1	1	$A + \overline{B} + \overline{C}$	M_3
1	0	0	$\overline{A} + B + C$	M_4
1	0	1	$\overline{A} + B + \overline{C}$	M_5
1	1	0	$\overline{A} + \overline{B} + C$	M_6
1	1	1	$\overline{A} + \overline{B} + \overline{C}$	M_7

6.3 Conversion from One Standard Form to Another

Recalling DeMorgan's theorem and comparing Tables 6.1 and 6.2, we observe that each maxterm is the complement of its corresponding minterm and vice versa.

Example 6.1

Prove that $\overline{M_1} = m_1$.

Proof:

From Table 6.2

$$M_1 = A + B + \overline{C}$$

Complementing M_1 and applying DeMorgan's theorem we get

$$\overline{M_1} = \overline{A + B + \overline{C}} = \overline{A}\overline{B}\overline{\overline{C}} = \overline{A}\overline{B}C$$

From Table 6.1

$$\overline{A}\overline{B}C = m_1 = \overline{M_1}$$

Example 6.2

Prove that $\overline{m_6} = M_6$.

Proof:

From Table 6.1

$$m_6 = AB\overline{C}$$

Complementing m_6 and applying DeMorgan's theorem we get

$$\overline{m_6} = \overline{AB\overline{C}} = \overline{A} + \overline{B} + \overline{\overline{C}} = \overline{A} + \overline{B} + C$$

From Table 6.2

$$\overline{A} + \overline{B} + C = M_6 = \overline{m_6}$$

6.4 Properties of Minterms and Maxterms

Two important properties of Boolean algebra state that:

1. Any Boolean function can be expressed as a *sum of minterms* where "sum" means the ORing of the terms.

1. Any Boolean function can be expressed as a *product of maxterms* where "product" means the ANDing of the terms.

Property 1 above implies that a Boolean function can be derived from a given truth table by assigning a minterm to each combination of variables that produces a logical 1 and then ORing all these terms.

Example 6.3

Using the truth table of Table 6.3, express the outputs* D and E as a sum of minterms.

TABLE 6.3 Truth table for Example 6.3

Inputs			Outputs	
X	Y	Z	D	B
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

---

* *In Chapter 7 we will see that this truth table represents a full subtractor where D is the difference and B is the borrow.*

Solution:

$$D = \overline{X}\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\overline{Z} + XYZ = m_1 + m_2 + m_4 + m_7 = \Sigma(1, 2, 4, 7) \quad (6.1)$$

$$B = \overline{X}\overline{Y}Z + \overline{X}Y\overline{Z} + \overline{X}YZ + XYZ = m_1 + m_2 + m_3 + m_7 = \Sigma(1, 2, 3, 7) \quad (6.2)$$

Property 2 above implies that a Boolean function can be derived from a given truth table by assigning a maxterm to each combination of variables that produces a logical 0 and then ANDing all these terms.

Example 6.4

Using the truth table of Table 6.3, express the outputs D and E as a product of maxterms.

Solution:

$$\begin{aligned} D &= (X + Y + Z)(X + \overline{Y} + \overline{Z})(\overline{X} + Y + \overline{Z})(\overline{X} + \overline{Y} + Z) \\ &= (M_0 \cdot M_3 \cdot M_5 \cdot M_6) = \Pi(0, 3, 5, 6) \end{aligned} \quad (6.3)$$

$$\begin{aligned} B &= (X + Y + Z)(\overline{X} + Y + Z)(\overline{X} + Y + \overline{Z})(\overline{X} + \overline{Y} + Z) \\ &= (M_0 \cdot M_4 \cdot M_5 \cdot M_6) = \Pi(0, 4, 5, 6) \end{aligned} \quad (6.4)$$

From Examples 6.3 and 6.4 we observe that we can convert a sum of minterms to a product of maxterms or vice versa if we replace the summation symbol Σ with the product symbol Π and we list the numbers missing from the original form. Thus, from relation (6.1), $\Sigma(1, 2, 4, 7)$ and the missing numbers within the parentheses appear in relation (6.3), i.e., $\Pi(0, 3, 5, 6)$. Likewise, from relation (6.2), $\Sigma(1, 2, 3, 7)$, and from relation (6.4), $\Pi(0, 4, 5, 6)$.

When a Boolean function is expressed in either a sum of minterms or a product of maxterms and each terms contains all variables, the function is said to be in *canonical form*. For example relations (6.1) through (6.4) are expressed in canonical form.

Most often, Boolean expressions consist of terms that do not contain all variables; for instance, in the expression $D = A\overline{B} + \overline{A}C + \overline{C}$ the variables C, B, and AB are missing from the first, second, and third terms respectively on the right side of this expression. However, it is possible to express such an expression as a sum of minterms by using appropriate postulates and theorems as it is illustrated by the following example.

Example 6.5

Express the function

$$D = f(A, B, C) = A + \overline{B}C \tag{6.5}$$

in a sum of minterms form.

Solution:

We observe that the variables B and C are missing from the first term, and the variable A is missing from the second term. From Chapter 5, Theorem 8(b), $A + \overline{A} = 1$, $B + \overline{B} = 1$ and thus the relation (6.5) can be written as

$$D = A(B + \overline{B}) + (A + \overline{A})\overline{B}C = AB + A\overline{B} + A\overline{B}C + \overline{A}\overline{B}C \tag{6.6}$$

We now observe that the variable C is missing from the first two terms on the right side of relation (6.6). Multiplication of those two terms by $C + \overline{C} = 1$ yields

$$\begin{aligned} D &= AB(C + \overline{C}) + A\overline{B}(C + \overline{C}) + A\overline{B}C + \overline{A}\overline{B}C \\ &= ABC + AB\overline{C} + A\overline{B}C + A\overline{B}\overline{C} + A\overline{B}C + \overline{A}\overline{B}C \end{aligned} \tag{6.7}$$

The third and fifth terms of the last expression above are the same; therefore, one can be deleted. Then,

$$\begin{aligned} D &= \overline{A}\overline{B}C + A\overline{B}\overline{C} + A\overline{B}C + AB\overline{C} + ABC \\ &= m_1 + m_4 + m_5 + m_6 + m_7 = \Sigma(1, 4, 5, 6, 7) \end{aligned}$$

To express a Boolean function as a product of maxterms, it must first be brought into a form of OR terms. This can be done with the use of the distributive law, Theorem 3(b) in Chapter 5, that is, $A + BC = (A + B)(A + C)$.

Example 6.6

Express the Boolean function

$$W = XY + \overline{X}Z \tag{6.8}$$

in a product of maxterms.

Solution:

We convert the given function into OR terms using the distributive law

$$A + BC = (A + B)(A + C)$$

in succession. Then,

$$W = XY + \overline{X}Z = (XY + \overline{X})(XY + Z) = (X + \overline{X})(\overline{X} + Y)(X + Z)(Y + Z)$$

and since $X + \overline{X} = 1$,

$$W = (\overline{X} + Y)(X + Z)(Y + Z)$$

We observe that each OR term is missing one variable. From Theorem 8(a) in Chapter 5, $A \cdot \overline{A} = 0$, and thus

$$(\overline{X} + Y) = (\overline{X} + Y + Z\overline{Z}) = (\overline{X} + Y + Z)(\overline{X} + Y + \overline{Z})$$

$$(X + Z) = (X + Y\overline{Y} + Z) = (X + Y + Z)(X + \overline{Y} + Z)$$

$$(Y + Z) = (X\overline{X} + Y + Z) = (X + Y + Z)(\overline{X} + Y + Z)$$

Removing those terms which appear twice, we get

$$\begin{aligned} W &= (X + Y + Z)(X + \overline{Y} + Z)(\overline{X} + Y + Z)(\overline{X} + Y + \overline{Z}) \\ &= M_0 \cdot M_2 \cdot M_4 \cdot M_5 = \Pi(0, 2, 4, 5) \end{aligned}$$

The canonical forms are very rarely the expressions with the least number of variables. A more common method of expressing a Boolean function is the *standard form* where the terms of the function may contain one, two, or any number of variables. There are two types of standard forms:

1. The *sum-of-products* (contrasted to sum of minterms) is a Boolean expression containing AND terms referred to as *product terms* of one or more variables each. The term "sum" implies the ORing of these terms. For example, the Boolean expression

   $$D = AB + \overline{A}B\overline{C} + \overline{B}$$

   is said to be expressed in sum-of-products form.

2. The *product-of-sums* (contrasted to product of maxterms) is a Boolean expression containing OR terms referred to as *sum terms* of one or more variable each. The term "product" implies the ANDing of these terms. For example, the Boolean expression

   $$E = (A + \overline{B} + C + \overline{D})(\overline{A} + \overline{C} + D)(B + D)$$

   is said to be expressed in a product-of-sums form.

Quite often, a Boolean function is expressed in a *non-standard form*. For example, the Boolean expression

$$Z = (VW + XY)(\overline{V}\,\overline{W} + \overline{X}\,\overline{Y})$$

is neither in a sum-of-products form, nor in a product-of-sums form because each term, i.e., XY

contains more that one variable. However, it can be converted to a standard form by using the distributive law to remove the parentheses, as illustrated by the following example.

Example 6.7

Express the Boolean function

$$Z = (VW + XY)(\overline{V}\,\overline{W} + \overline{X}\,\overline{Y})$$

in a sum-of-products form

Solution:

Using the distributive law, we get

$$Z = (VW + XY)(\overline{V}\,\overline{W} + \overline{X}\,\overline{Y}) = VW\overline{V}\,\overline{W} + VW\overline{X}\,\overline{Y} + \overline{V}\,\overline{W}XY + XY\overline{X}\,\overline{Y}$$

From Theorem 8(a), Chapter 5, $A \cdot \overline{A} = 0$, and thus the first and fourth terms in the expression above are zero. Hence,

$$Z = VW\overline{X}\,\overline{Y} + \overline{V}\,\overline{W}XY$$

and Z is now expressed in a sum-of-products form.

6.5 Summary

- The possible combinations of a set of variables defines a distinct area in a Vern diagram.
- For n variables there are 2^n minterms designated as m_j where the subscript j represents the decimal equivalent of the binary number of the minterm. Each minterm is obtained from an ANDed term of n variables with each variable being complemented if the corresponding bit is a logical 0, and uncomplemented if the corresponding bit is logical 1.
- For n variables there are 2^n maxterms designated as M_j where the subscript j represents the decimal equivalent of the binary number of the maxterm. Each maxterm is composed of two or more variables or their complements ORed together with each variable being complemented if the corresponding bit is a logical 1, and uncomplemented if the corresponding bit is logical 0.
- Each maxterm is the complement of its corresponding minterm and vice versa.
- Any Boolean function can be expressed as a sum of minterms where "sum" means the ORing of the terms.
- Any Boolean function can be expressed as a product of maxterms where "product" means the ANDing of the terms.
- We can convert a sum of minterms to a product of maxterms or vice versa if we replace the summation symbol Σ with the product symbol Π and we list the numbers missing from the original form.
- When a Boolean function is expressed in either a sum of minterms or a product of maxterms and each terms contains all variables, the function is said to be in canonical form.
- Any Boolean function that does not contain all variables can be expressed in a sum of minterms form by using appropriate postulates and theorems.
- To express a Boolean function as a product of maxterms, we must first convert it into a form of OR terms. This can be done with the use of the distributive law, Theorem 3(b) in Chapter 5, that is, $A + BC = (A + B)(A + C)$.
- Most Boolean functions are expressed in standard form either in a sum-of-products form or in a product-of-sums form.
- If a Boolean function is expressed in a non-standard form, it can be converted to a standard form by using the distributive law.

6.6 Exercises

1. The truth table of Example 6.3 is repeated below for convenience.

TABLE 6.4 Truth table for Exercises 6.1 and 6.2

Inputs			Outputs	
X	Y	Z	D	B
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

a. Derive an expression for the complement of D as a sum of minterms, i.e,

$$\overline{D} = f(X, Y, Z) = \Sigma(m_i, m_j, m_k, \ldots)$$

b. Complement the expression for $\overline{D}$ and using DeMorgan's theorems derive an expression as a product of maxterms. Compare this expression with that of relation (6.3) of Example 6.3.

2. Using Table 6.4 above

a. Derive an expression for the complement of B as a sum of minterms, i.e,

$$\overline{B} = f(X, Y, Z) = \Sigma(m_i, m_j, m_k, \ldots)$$

b. Complement the expression for $\overline{B}$ and using DeMorgan's theorems derive an expression as a product of maxterms. Compare this expression with that of relation (6.4) of Example 6.3.

3. Fill-in the missing data for the blank rows of the table below.

Variables				Minterms		Maxterms	
A	B	C	D	Term	Designation	Term	Designation
0	0	0	0	$\overline{A} \cdot \overline{B} \cdot \overline{C} \cdot \overline{D}$	m_0	$A + B + C + D$	M_0
0	0	0	1	$\overline{A} \cdot \overline{B} \cdot \overline{C} \cdot D$	m_1	$A + B + C + \overline{D}$	M_1
...	...	...	...	...	...	...	...
1	1	1	1	$A \cdot B \cdot C \cdot D$	m_{15}	$\overline{A} + \overline{B} + \overline{C} + \overline{D}$	M_{15}

4. Express the Boolean function

$$D = (\overline{A} + B)(\overline{B} + C)$$

a. as a product of maxterms.

b. as a sum of minterms

5. Express the Boolean function

$$Z = (VW + XY)(\overline{V}\,\overline{W} + \overline{X}\,\overline{Y})$$

in a product-of-sums form

6.7 Solutions to End-of-Chapter Exercises

1.

a. From Table 6.4,

$$\overline{D} = \overline{X}\overline{Y}\overline{Z} + \overline{X}YZ + X\overline{Y}Z + XY\overline{Z} = m_0 + m_3 + m_5 + m_6 = \Sigma(0, 3, 5, 6)$$

b.

$$\overline{\overline{D}} = D = \overline{\overline{X}\overline{Y}\overline{Z} + \overline{X}YZ + X\overline{Y}Z + XY\overline{Z}} = \overline{\overline{X}\overline{Y}\overline{Z}} \cdot \overline{\overline{X}YZ} \cdot \overline{X\overline{Y}Z} \cdot \overline{XY\overline{Z}}$$

$$= (X + Y + Z) \cdot (X + \overline{Y} + \overline{Z}) \cdot (\overline{X} + Y + \overline{Z}) \cdot (\overline{X} + \overline{Y} + Z)$$

$$= M_0 \cdot M_3 \cdot M_5 \cdot M_6 = \Pi(0, 3, 5, 6)$$

We observe that the above expression is the same as relation (6.3) of Example 6.3.

2.

a. From Table 6.4,

$$\overline{B} = \overline{X}\overline{Y}\overline{Z} + X\overline{Y}\overline{Z} + X\overline{Y}Z + XY\overline{Z} = m_0 + m_4 + m_5 + m_6 = \Sigma(0, 4, 5, 6)$$

b.

$$\overline{\overline{B}} = B = \overline{\overline{X}\overline{Y}\overline{Z} + X\overline{Y}\overline{Z} + X\overline{Y}Z + XY\overline{Z}} = \overline{\overline{X}\overline{Y}\overline{Z}} \cdot \overline{X\overline{Y}\overline{Z}} \cdot \overline{X\overline{Y}Z} \cdot \overline{XY\overline{Z}}$$

$$= (X + Y + Z) \cdot (\overline{X} + Y + Z) \cdot (\overline{X} + Y + \overline{Z}) \cdot (\overline{X} + \overline{Y} + Z)$$

$$= M_0 \cdot M_4 \cdot M_5 \cdot M_6 = \Pi(0, 4, 5, 6)$$

We observe that the above expression is the same as relation (6.4) of Example 6.3.

3.

Variables				Minterms		Maxterms	
A	B	C	D	Term	Designation	Term	Designation
0	0	0	0	$\overline{A} \cdot \overline{B} \cdot \overline{C} \cdot \overline{D}$	m_0	$A + B + C + D$	M_0
0	0	0	1	$\overline{A} \cdot \overline{B} \cdot \overline{C} \cdot D$	m_1	$A + B + C + \overline{D}$	M_1
0	0	1	0	$\overline{A} \cdot \overline{B} \cdot C \cdot \overline{D}$	m_2	$A + B + \overline{C} + D$	M_2
0	0	1	1	$\overline{A} \cdot \overline{B} \cdot C \cdot D$	m_3	$A + B + \overline{C} + \overline{D}$	M_3
0	1	0	0	$\overline{A} \cdot B \cdot \overline{C} \cdot \overline{D}$	m_4	$A + \overline{B} + C + D$	M_4
0	1	0	1	$\overline{A} \cdot B \cdot \overline{C} \cdot D$	m_5	$A + \overline{B} + C + \overline{D}$	M_5
0	1	1	0	$\overline{A} \cdot B \cdot C \cdot \overline{D}$	m_6	$A + \overline{B} + \overline{C} + D$	M_6
0	1	1	1	$\overline{A} \cdot B \cdot C \cdot D$	m_7	$A + \overline{B} + \overline{C} + \overline{D}$	M_7
1	0	0	0	$A \cdot \overline{B} \cdot \overline{C} \cdot \overline{D}$	m_8	$\overline{A} + B + C + D$	M_8
1	0	0	1	$A \cdot \overline{B} \cdot \overline{C} \cdot D$	m_9	$\overline{A} + B + C + \overline{D}$	M_9
1	0	1	0	$A \cdot \overline{B} \cdot C \cdot \overline{D}$	m_{10}	$\overline{A} + B + \overline{C} + D$	M_{10}
1	0	1	1	$A \cdot \overline{B} \cdot C \cdot D$	m_{11}	$\overline{A} + B + \overline{C} + \overline{D}$	M_{11}
1	1	0	0	$A \cdot B \cdot \overline{C} \cdot \overline{D}$	m_{12}	$\overline{A} + \overline{B} + C + D$	M_{12}
1	1	0	1	$A \cdot B \cdot \overline{C} \cdot D$	m_{13}	$\overline{A} + \overline{B} + C + \overline{D}$	M_{13}
1	1	1	0	$A \cdot B \cdot C \cdot \overline{D}$	m_{14}	$\overline{A} + \overline{B} + \overline{C} + D$	M_{14}
1	1	1	1	$A \cdot B \cdot C \cdot D$	m_{15}	$\overline{A} + \overline{B} + \overline{C} + \overline{D}$	M_{15}

4.

a.

From Chapter 5, Theorem 8(b), $A + \overline{A} = 1$. Multiplication of the first term by $C + \overline{C} = 1$, the second term by $A + \overline{A} = 1$, and using the distributive law, Theorem 3(b) in Chapter 5, that is, $A + BC = (A + B)(A + C)$ we get

$$\begin{aligned} D &= (\overline{A} + B)(C + \overline{C})(\overline{B} + C)(A + \overline{A}) \\ &= (A + \overline{B} + C)(\overline{A} + B + C)(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + C) \\ &= M_2 \cdot M_4 \cdot M_5 \cdot M_6 = \Pi(2, 4, 5, 6) \end{aligned}$$

b.

As we've learned, we can convert a product of maxterms to a sum of minterms if we replace the product symbol Π with the summation symbol Σ and list the numbers missing from the original form. From part (a) $\Pi(2, 4, 5, 6)$, and the missing numbers are 0, 1, 3, and 7. Therefore, in a sum of minterms form,

$$D = \Sigma(0, 1, 3, 7) = m_0 + m_1 + m_3 + m_7 = \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C + \overline{A}BC + ABC$$

5.

By the distributive law, $A + BC = (A + B)(A + C)$. Thus,

$$VW + XY = (VW + X)(VW + Y) = (X + V)(X + W)(Y + V)(Y + W)$$
$$\overline{V}\,\overline{W} + \overline{X}\,\overline{Y} = (\overline{V}\,\overline{W} + \overline{X})(\overline{V}\,\overline{W} + \overline{Y}) = (\overline{X} + \overline{V})(\overline{X} + \overline{W})(\overline{Y} + \overline{V})(\overline{Y} + \overline{W})$$

Then, in a product-of-sums form,

$$Z = (X + V)(X + W)(Y + V)(Y + W)(\overline{X} + \overline{V})(\overline{X} + \overline{W})(\overline{Y} + \overline{V})(\overline{Y} + \overline{W})$$

Chapter 7

Combinational Logic Circuits

This chapter is an introduction to combinational logic circuits. These are logic circuits where the output(s) depend solely on the input(s). It begins with methods of implementation of logic diagrams from Boolean expressions, the derivation of Boolean expressions from logic diagrams, input and output waveforms, and Karnaugh maps.

7.1 Implementation of Logic Diagrams from Boolean Expressions

In this section we will present procedures for converting Boolean expressions to logic diagrams using a specific type of gates or a combination of gates. The procedure is best illustrated with the examples that follow.

Important: The ANDing operation has precedence over the ORing operation. For instance, the Boolean expression $AB + C$ implies that A must first be ANDed with B and the result must be ORed with C as shown in Figure 7.1.

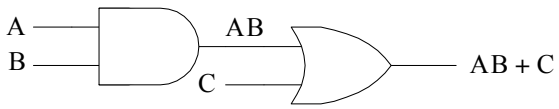


Figure 7.1. Implementation of the Boolean expression $AB + C$

Note: In all examples that follow, we will assume that the variables are available only in the uncomplemented state.

Example 7.1

Implement the complementation (inversion) operation A to $\overline{A}$ with a NAND gate, and B to $\overline{B}$ with a NOR gate.

Solution:

As shown in Figure 7.2, we can perform the inversion operation with either a NAND gate or a NOR gate by tying the inputs together.

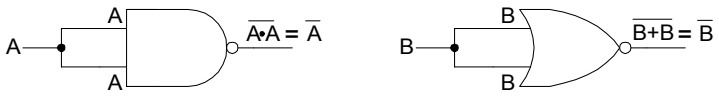


Figure 7.2. Using NAND and NOR gates for the inversion operation

Example 7.2

Implement DeMorgan's theorem $\overline{A \cdot B} = \overline{A} + \overline{B}$ with

a. a NAND gate

b. an OR gate and Inverters

Solution:

The implementations are shown in Figure 7.3 where the bubbles on the lower right OR gate imply inversion. By DeMorgan's theorem the gates on the right side are equivalent, and thus we conclude that a 2-input OR gate preceded by two inverters can be replaced by a single 2-input NAND gate.

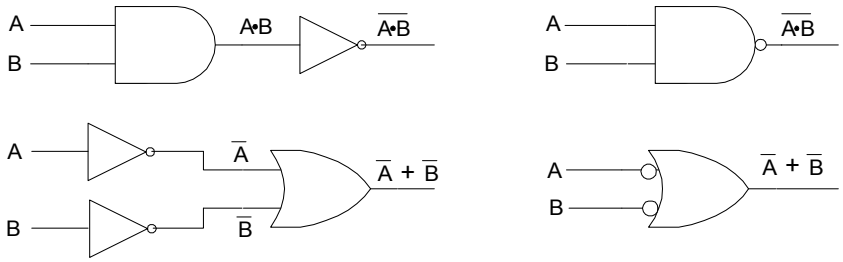


Figure 7.3. Implementation of DeMorgan's theorem $\overline{A \cdot B} = \overline{A} + \overline{B}$

Example 7.3

Implement the XOR function $A \oplus B = A \cdot \overline{B} + \overline{A} \cdot B$ * with

a. A combination of AND gates, OR gates, and Inverters

b. NAND gates only

Solution:

a.

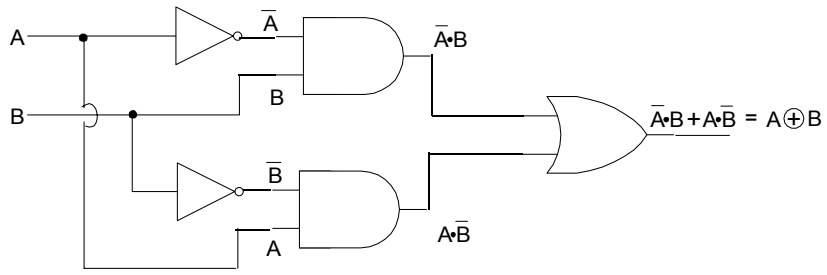


Figure 7.4. Implementation of the XOR function with AND gates, OR gate, and Inverters

* *Whereas* $A + B$ *implies the OR operation,* $A \oplus B$ *implies the Exclusive-OR operation*

This implementation requires three ICs, i.e., an SN7404, an SN7408, and an SN7432.

b.

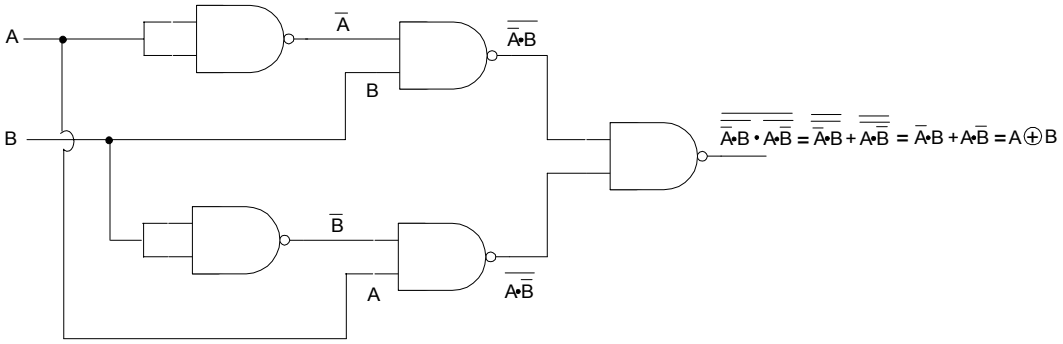

A
B
$\overline{A}$
$\overline{A \cdot B}$
B
$\overline{B}$
A
$\overline{A \cdot \overline{B}}$
$\overline{\overline{A \cdot B} \cdot \overline{A \cdot \overline{B}}} = \overline{\overline{A \cdot B}} + \overline{\overline{A \cdot \overline{B}}} = \overline{A} \cdot B + A \cdot \overline{B} = A \oplus B$

Figure 7.5. Implementation of the XOR function with NAND gates only

This implementation requires only two IC SN7400 devices.

Example 7.4

Implement the Boolean expression $D = A(\overline{B}\overline{C} + BC) + \overline{A}(B\overline{C} + \overline{B}C)$ with XOR gates only.

Solution:

The Boolean expression $B\overline{C} + \overline{B}C$ represents the XOR operation and the expression $\overline{B}\overline{C} + BC$ represents the XNOR operation. Since the XNOR operation is the complement of the XOR operation, we make the substitution

$$\overline{B}\overline{C} + BC = \overline{B\overline{C} + \overline{B}C}$$

and thus

$$D = A(\overline{B\overline{C} + \overline{B}C}) + \overline{A}(B\overline{C} + \overline{B}C) = A \oplus B \oplus C$$

This expression is implemented with two XOR gates as shown in Figure 7.6.

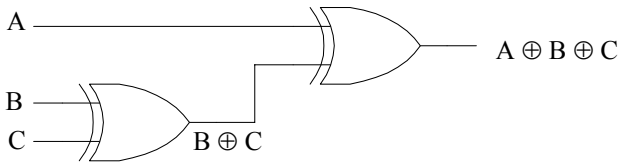


Figure 7.6. Implementation of the Boolean expression $D = A(\overline{B}\overline{C} + BC) + \overline{A}(B\overline{C} + \overline{B}C)$ *with XOR gates*

Thus, the given expression can be implemented with one SN7486 device.

The internal construction of the NAND and NOR gates* is simpler than those of the AND and OR gates and thus it is sometimes preferable to use NAND or NOR gates to implement Boolean expressions.

Example 7.5

Using NAND gates only implement

a. The AND operation X = ABC

b. The OR operation Y = A + B + C

Solution:

a.

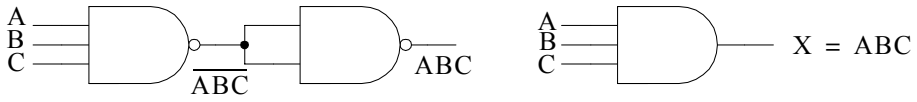


Figure 7.7. Implementation of the AND operation with NAND gates

b.

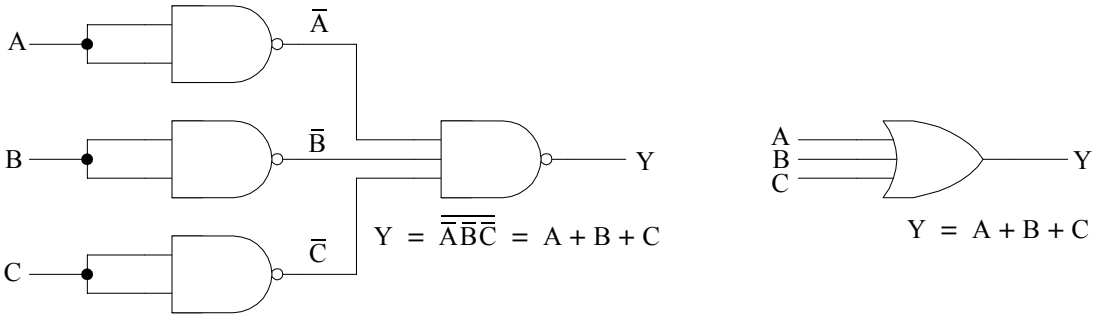


Figure 7.8. Implementation of the OR operation with NAND gates

---

Example 7.6

Using NOR gates only implement

a. The AND operation X = ABC

b. The OR operation Y = A + B + C

---

* *For a detailed discussion on the internal construction of logic gates, please refer to Electronic Devices and Amplifier Circuits, ISBN 0-9744239-4-7.*

Solution:

a.

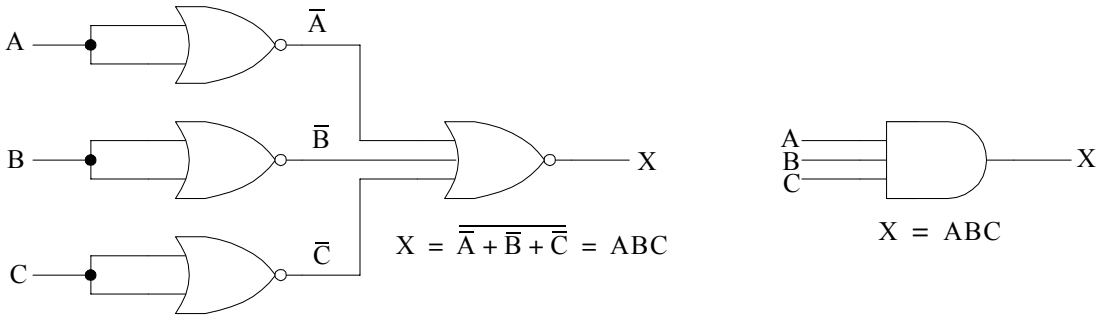


Figure 7.9. Implementation of the AND operation with NOR gates

b.

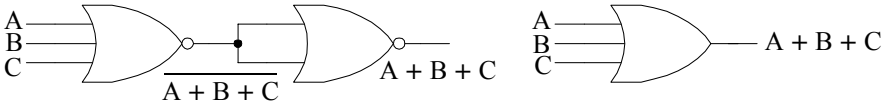


Figure 7.10. Implementation of the OR operation with NOR gates

The results of Examples 7.5 and 7.6 can be summarized with the sketch of Figure 7.11.

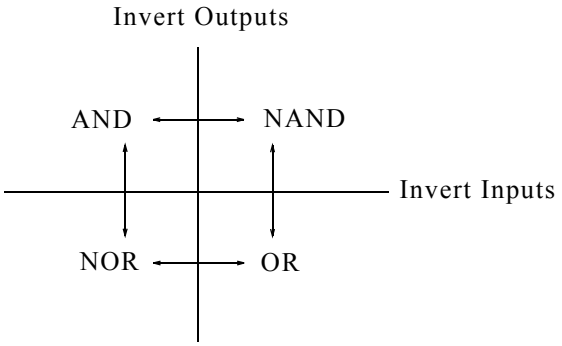


Figure 7.11. Conversion among AND, OR, NAND, and NOR gates

Figure 7.11 shows that we can replace an AND gate with a NAND gate by inverting the output of the AND gate. We observe that the conversion is bi-directional, that is, we can replace a NAND gate with an AND gate by inverting the output of the NAND gate. Similarly, we can replace an OR gate with a NAND gate by inverting the inputs of the OR gate. To replace an AND gate with an OR gate, we must invert both inputs and outputs.

Example 7.7

Using NAND gates only implement the Boolean expression $W = X(\overline{Y} + Z) + \overline{X}Y$

Solution:

It is convenient to start the implementation with AND gates, OR gates, and Inverters as shown in Figure 7.12.

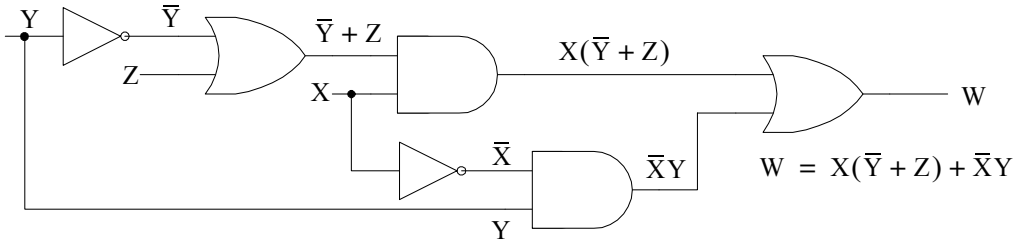


Figure 7.12. Implementation of $W = X(\overline{Y} + Z) + \overline{X}Y$ *with AND gates, OR gates, and Inverters*

Using the sketch of Figure 7.11, we obtain the logic diagram of Figure 7.13.

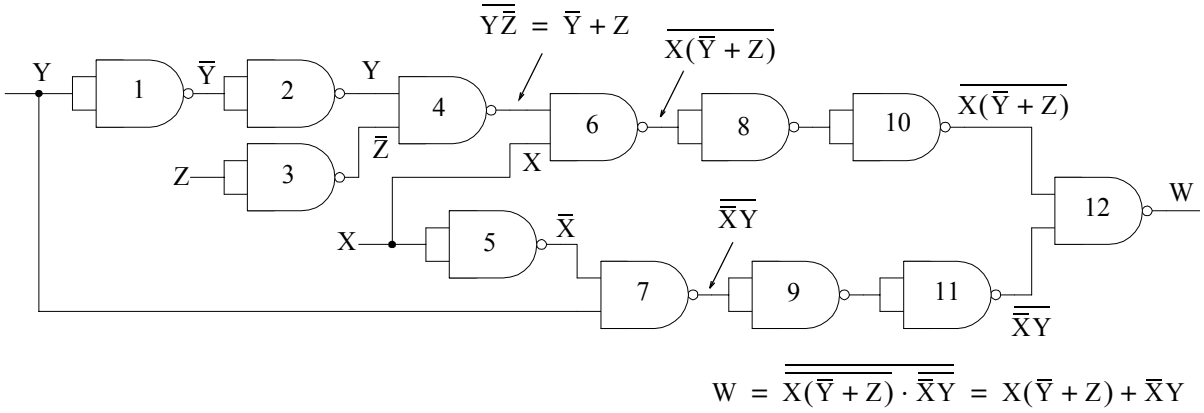
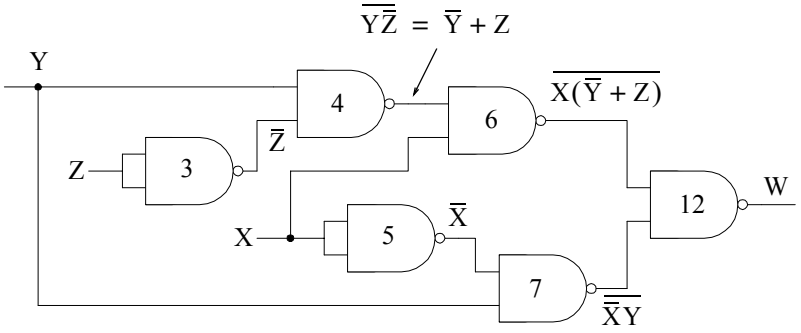


Figure 7.13. Implementation of $W = X(\overline{Y} + Z) + \overline{X}Y$ *with NAND gates, unsimplified*

From Figure 7.13 we observe that gates 1 and 2, 8 and 10, and 9 and 11 can be eliminated since they perform two successive inversions. Therefore, the logic circuit reduces to the one shown in Figure 7.14.



$$W = \overline{\overline{X(\bar{Y} + Z)} \cdot \overline{\bar{X}Y}} = X(\bar{Y} + Z) + \bar{X}Y$$

Figure 7.14. Implementation of $W = X(\bar{Y} + Z) + \bar{X}Y$ *with NAND gates, simplified*

Example 7.8

Using NOR gates only implement the Boolean expression $D = A\bar{B}C + \bar{A}B\bar{C} + AB$

Solution:

It is convenient to start the implementation with AND gates, OR gates, and Inverters as shown in Figure 7.15.

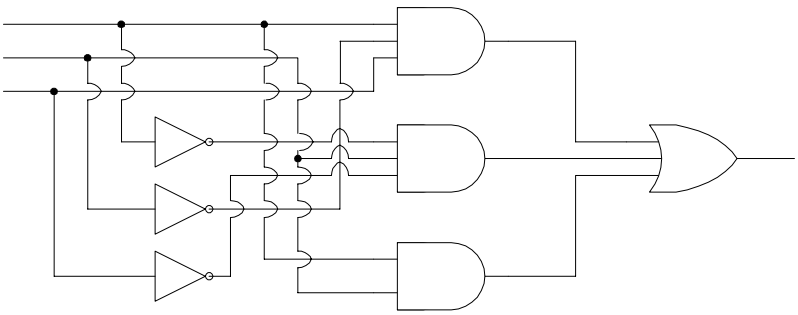


Figure 7.15. Implementation of $D = A\bar{B}C + \bar{A}B\bar{C} + AB$ *with AND gates, OR gates, and Inverters*

From Figure 7.16 we observe that gates 1 and 2, 3 and 4, and 5 and 6 can be eliminated since they perform two successive inversions. Therefore, the logic circuit reduces to the one shown in Figure 7.17.

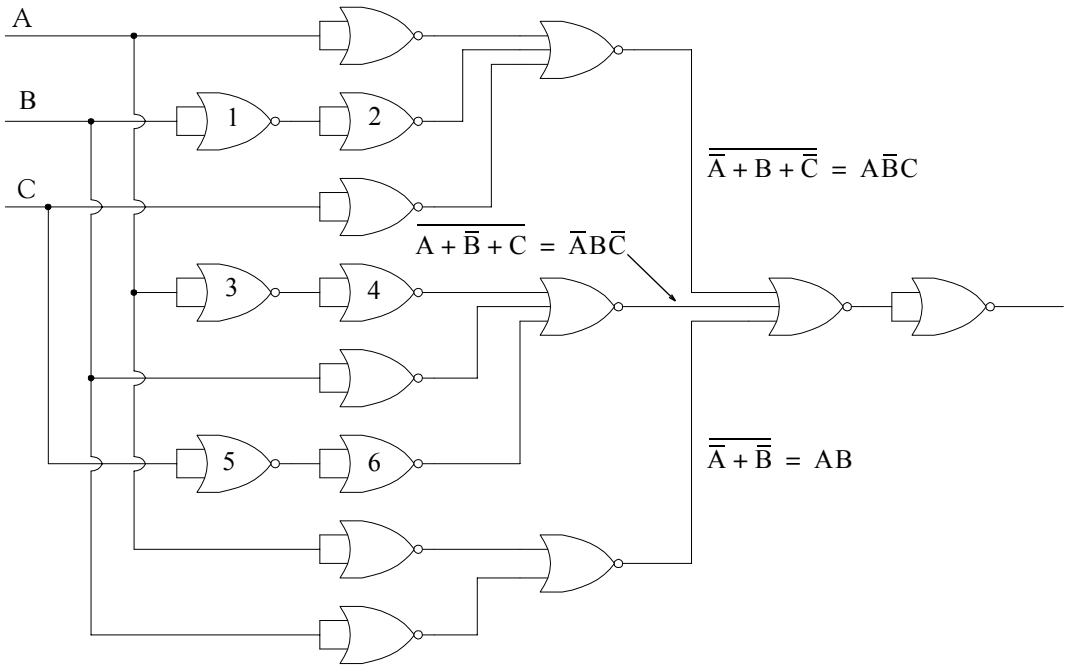


Figure 7.16. Implementation of $D = A\overline{B}C + \overline{A}B\overline{C} + AB$ *with NOR gates - unsimplified*

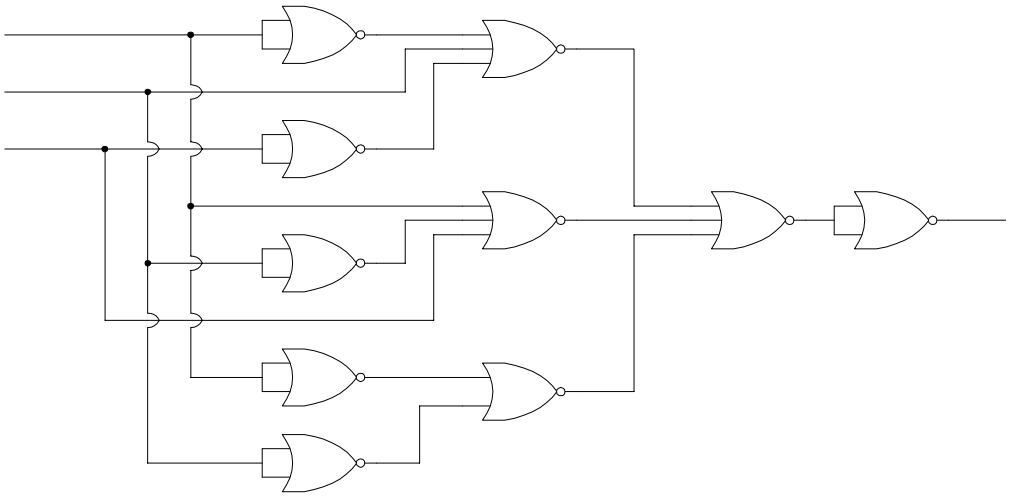


Figure 7.17. Implementation of $D = A\overline{B}C + \overline{A}B\overline{C} + AB$ *with NOR gates - simplified*

Often, it is possible to simplify a Boolean expression before implementing the logic diagram as it is illustrated by the following example.

Example 7.9

First simplify, then implement the Boolean expression $D = \overline{A}BC + A\overline{B}C + ABC + B\overline{C}$.

Solution:

Since $X + X = X$, we can add the term ABC to the given expression and using theorems 3.a, 8.b, and 7.b in Appendix E, we get

$$\begin{aligned} D &= \overline{A}BC + ABC + A\overline{B}C + ABC + B\overline{C} \\ &= BC(A + \overline{A}) + AC(B + \overline{B}) + B\overline{C} \\ &= BC + AC + B\overline{C} \\ &= B(C + \overline{C}) + AC \\ &= B + AC \end{aligned}$$

The simplified expression can be implemented as shown in Figure 7.18.

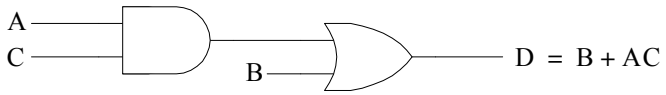


Figure 7.18. Implementation of the simplified expression $D = B + AC$ *for Example 7.9*

7.2 Obtaining Boolean Expressions from Logic Diagrams

When a logic circuit is given, the Boolean expression describing that logic circuit can be obtained by combining the input variables in accordance with the logic gate functions. The procedure is best illustrated with the examples that follow.

Example 7.10

For the logic circuit of Figure 7.19 find $D = f(A, B, C)$, that is, express the output D in terms of the inputs A, B, and C. If possible, simplify the Boolean expression obtained, and implement it with a simplified logic diagram.

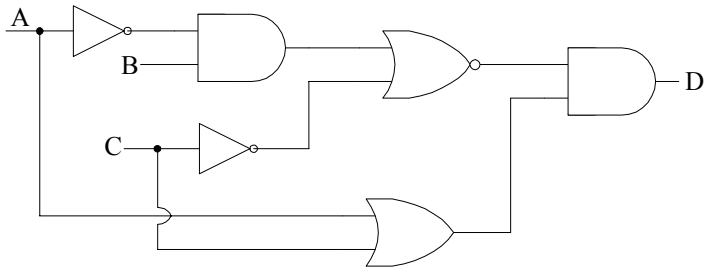


Figure 7.19. Logic diagram for Example 7.10

Solution:

We label the output of each gate as shown in Figure 7.20.

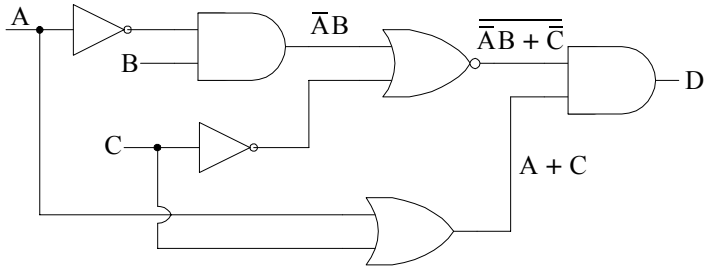


Figure 7.20. Outputs of the gates of logic diagram for Example 7.10

From Figure 7.20

$$D = (\overline{\overline{A}B + \overline{C}})(A + C) = (\overline{\overline{A}B}C)(A + C) = [(A + \overline{B})C](A + C)$$
$$= (A + \overline{B})(AC + CC) = (A + \overline{B})(AC + C) = AAC + AC + A\overline{B}C + \overline{B}C$$
$$= AC + AC + \overline{B}C(A + 1) = AC + \overline{B}C(1) = AC + \overline{B}C$$
$$= (A + \overline{B})C$$

The simplified logic diagram is shown in Figure 7.21.

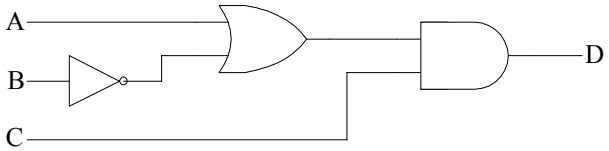


Figure 7.21. Simplified logic diagram for Example 7.10

7.3 Input and Output Waveforms

In a practical logic circuit the inputs, and consequently the outputs of the gates comprising the logic circuit change state with time and it is convenient to represent the changing inputs and outputs by waveforms referred to as *timing diagrams* such as that shown in Figure 7.22.*

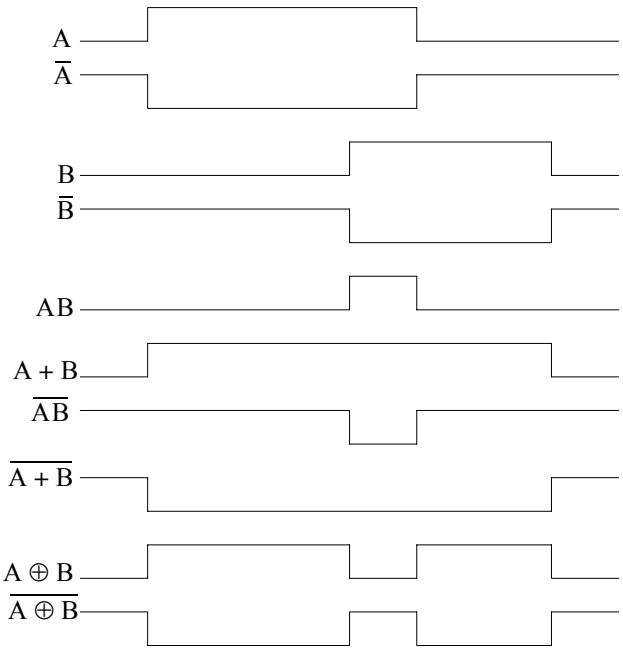


Figure 7.22. Typical timing diagrams

In the timing diagrams of Figure 7.22 it is assumed that the first two represent the inputs are A and B and those below represent their complements, ANDing, ORing, NANDing, NORing, XORing, and XNORing.

Example 7.11

For the logic diagram of Figure 7.23(a), the inputs A and B vary with time as shown in Figure 7.23(b). Sketch the timing diagram for the output C in the time interval $T_1 \le T \le T_2$.

---

* *For convenience, we have assumed that the changes are instantaneous but in reality there is always a delay during the changing intervals and these are referred to as rising and falling times.*

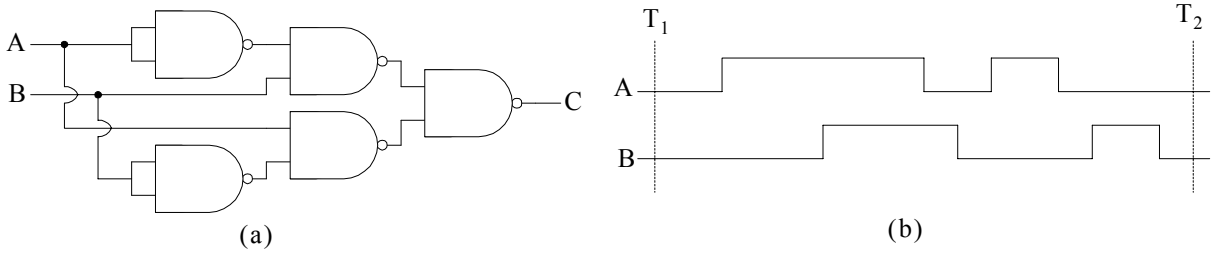


Figure 7.23. Logic and timing diagrams for Example 7.11

Solution:

From the logic diagram of Figure 7.23(a), $C = \overline{\overline{A}B}\,\overline{A\overline{B}} = \overline{A}B + A\overline{B} = A \oplus B$, that is, the logic diagram represents an XOR gate implemented with NAND gates. The timing diagram for the output C is shown in Figure 7.24.

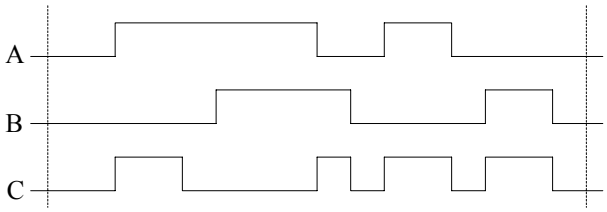


Figure 7.24. Timing diagrams for inputs and output of the logic diagram of Figure 7.23(a)

7.4 Karnaugh Maps

A *Karnaugh map*, henceforth referred to as *K-map*, is a matrix of squares. In general, a Boolean expression with *n* variables can be represented by a K-map of 2^n squares where each square represents a row of an equivalent truth table. A K-map provides a very powerful method of reducing Boolean expressions to their simplest forms.

7.4.1 K-map of Two Variables

Figure 7.25 shows a two-variable K-map with four squares where each square represents one combination of the variables.

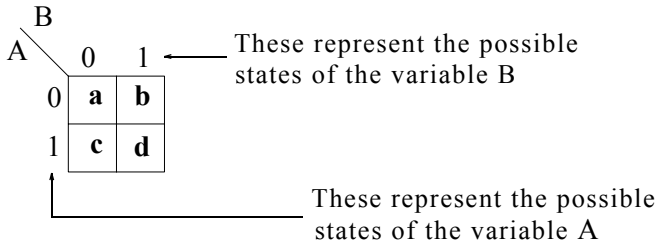


Figure 7.25. K-map of two variables

In figure 7.25,

square **a** represents the combination $A = 0$ and $B = 0$, that is, $\mathbf{a} \Rightarrow \overline{A}\overline{B}$

square **b** represents the combination $A = 0$ and $B = 1$, that is, $\mathbf{b} \Rightarrow \overline{A}B$

square **c** represents the combination $A = 1$ and $B = 0$, that is, $\mathbf{c} \Rightarrow A\overline{B}$

square **d** represents the combination $A = 1$ and $B = 1$, that is, $\mathbf{d} \Rightarrow AB$

For example, the Boolean expression $C = \overline{A}B + A\overline{B}$ can be shown in a K-map as indicated in Figure 7.26 where the 1s denote the conditions for which the Boolean expression is true (logical 1).

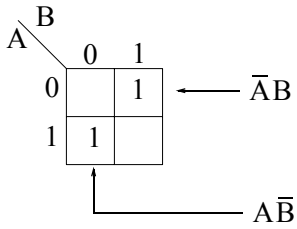


Figure 7.26. K-map for the Boolean expression $C = \overline{A}B + A\overline{B}$

For simplicity, we enter only 1s1 in the squares of a K-map and it is understood that all other empty squares contain 0s.

We can also derive a Boolean expression from a K-map. For example, from the K-map shown in Figure 7.27 we derive the Boolean expression $Z = \overline{X}\overline{Y} + XY$

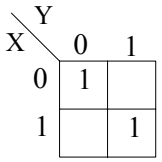


Figure 7.27. K-map for the Boolean expression $Z = \overline{X}\overline{Y} + XY$

7.4.2 K-map of Three Variables

A Boolean expression with three variables can be represented by a K-map with $2^3 = 8$ squares as shown in Figure 7.28.

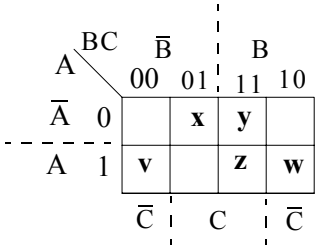


Figure 7.28. K-map of three variables

K-rnaps are arranged so that adjacent squares represent terms which differ by only one variable which appears in one square as a complemented variable and as an uncomplemented variable in an adjacent square. In Figure 7.28 we observe that the order of the combinations of the variables B and C are in the sequence 00, 01, 11, and 10. This arrangement allows only one variable change between terms in adjacent squares – terms next to each other row-wise or next to each other column-wise. Adjacent squares are also those of the leftmost columns and the squares of the rightmost column since they differ in only one variable. The same is true for the top and bottom rows.Terms in a diagonal arrangement are not considered adjacent squares. For example, squares **x** and **y**, **y** and **z**, **z** and **w**, and **v** and **w** are adjacent to each other. However, the diagonal squares **v** and **x**, **x** and **z**, and **y** and **w** are not adjacent to each other.

7.4.3 K-map of Four Variables

A Boolean expression with four variables can be represented by a K-map with $2^4 = 16$ squares as shown in Figure 7.29.

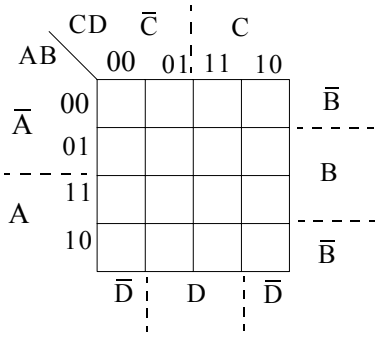


Figure 7.29. K-map of four variables

This arrangement shows again that K-rnaps are arranged so that adjacent squares represent terms that differ by only one variable which appears in one square as a complemented variable

and as an uncomplemented variable in an adjacent square. Consider, for example, the Boolean expression

$$E = \overline{A}BCD + ABCD$$

where the two terms of the right side differ by only one variable (in this case A) which is complemented in the first term but it is uncomplemented in the second term. These terms are shown as 1s in the K-map of Figure 7.30.

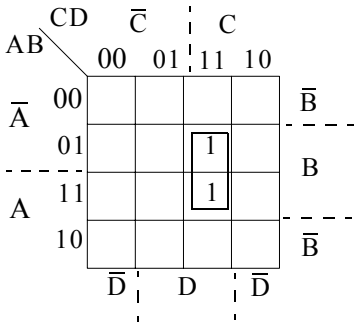


Figure 7.30. K-map of for the Boolean expression $E = \overline{A}BCD + ABCD$

The K-map of Figure 7.30 map indicates that the two squares with 1s are positioned in an area described by BCD, and thus the terms $\overline{A}BCD$ and ABCD can be combined into the single term BCD. The same result is, of course, obtained by application of one of the Boolean algebra theorems as

$$E = \overline{A}BCD + ABCD = (A + \overline{A})BCD = BCD$$

but the procedure with K-maps is more practical and more efficient wherever a Boolean expression contains more than just two terms.

The above example indicates that two terms each containing four variables may be replaced by a single term involving only three variables if these terms are positioned in adjacent squares of a K-map. In general, any pair of adjacent squares can be replaced by a single term containing one variable less than the two individual terms. Furthermore, if 2^n squares are adjacent, they can combine to produce a single term from which n variables have been eliminated.

As mentioned earlier, adjacent squares are also those of the leftmost columns and the squares of the rightmost column since they differ in only one variable. The same is true for the top and bottom rows. We may think of a K-map as being folded horizontally or vertically so that the edges are placed next to each other. The four corners of a K-map are also adjacent to each other.

Example 7.12

Use a K-map to simplify the Boolean expression

$$E = \overline{A}\overline{B}CD + AB\overline{C}\overline{D} + ABC\overline{D} + A\overline{B}CD$$

Solution:

This expression contains four variables and thus can be mapped into a K-map of $2^4 = 16$ squares as shown in Figure 7.31.

The left and right columns are adjacent to each other and thus the terms $AB\overline{C}\overline{D}$ and $ABC\overline{D}$ can be combined to yield the single term $AB\overline{D}$. Likewise, the top and bottom rows are adjacent to each other and thus the terms $\overline{A}\overline{B}CD$ and $A\overline{B}CD$ can be combined to yield the single term $\overline{B}CD$. Therefore, the given expression can be written in simplified form as

$$E = AB\overline{D} + \overline{B}CD$$

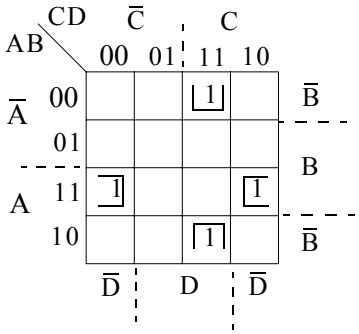


Figure 7.31. K-map for the Boolean expression $E = \overline{A}\overline{B}CD + AB\overline{C}\overline{D} + ABC\overline{D} + A\overline{B}CD$

7.4.4 General Procedures for Using a K-map of n Squares

To use a K-map effectively, we must follow the procedures below.

1. Each square with a 1 in it must be included at least once in the combinations of the squares which were selected. However, a particular square may be included in more than one combination of squares if it can be combined with other squares.
2. The number of squares to be combined must be a power of 2, i.e., squares can be combined in groups of two, four, eight, and so on squares.
3. Each combination should be selected to include the highest possible number of squares.

The following examples illustrate the procedures.

Example 7.13

Use a K-map to simplify the Boolean expression

$$D = \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + A\overline{B}\overline{C} + AB\overline{C}$$

Solution:

This expression contains three variables and thus can be mapped into a K-map of $2^3 = 8$ squares as shown in Figure 7.32.

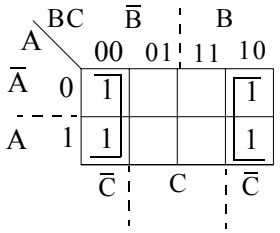


Figure 7.32. K-map for Example 7.13

The K-map of Figure 7.32 shows four adjacent squares. Therefore, $4 = 2^n$ or $n = 2$ and this indicates that two variables can be eliminated. Accordingly, the given Boolean expression simplifies to

$$D = \overline{C}$$

Example 7.14

A Boolean expression is mapped into the K-map of Figure 7.33. Derive the simplest Boolean expression $E = f(A, B, C, D)$.

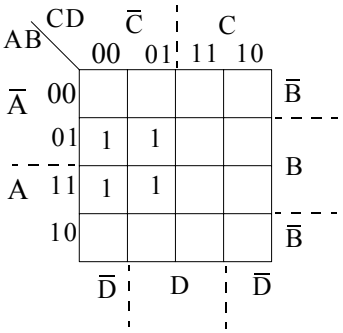


Figure 7.33. K-map for Example 7.14

Solution:

The K-map of Figure 7.33 shows four adjacent squares and these can be grouped as shown in Figure 7.34.

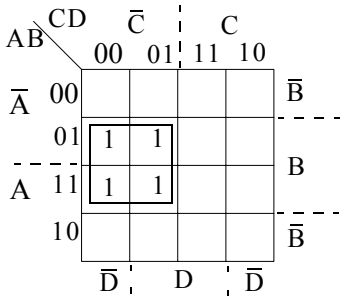


Figure 7.34. K-map for Example 7.14 with grouped variables

Therefore, $4 = 2^n$ or $n = 2$ and this indicates that two variables can be eliminated. Accordingly, the simplest Boolean expression is

$$E = B\overline{C}$$

Example 7.15

A Boolean expression is mapped into the K-map of Figure 7.35. Derive the simplest Boolean expression $E = f(A, B, C, D)$.

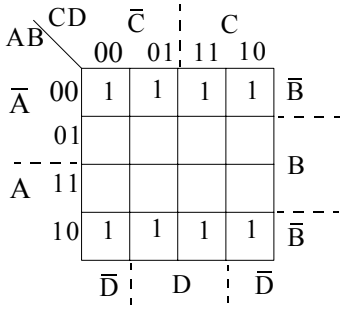


Figure 7.35. K-map for Example 7.15

Solution:

The K-map of Figure 7.35 shows four adjacent squares and these can be grouped as shown in Figure 7.36.

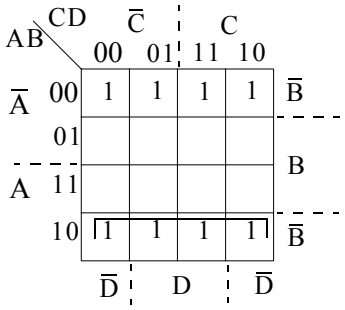


Figure 7.36. K-map for Example 7.15 with grouped variables

Therefore, $8 = 2^n$ or $n = 3$ and this indicates that three variables can be eliminated. Accordingly, the simplest Boolean expression is

$$E = \overline{B}$$

Example 7.16

Use a K-map to simplify the Boolean expression

$$E = \overline{A}\overline{B}\overline{C}D + \overline{A}CD + \overline{A}\overline{C} + C$$

Solution:

This expression contains four variables and thus can be mapped into a K-map of $2^4 = 16$ squares as shown in Figure 7.37.

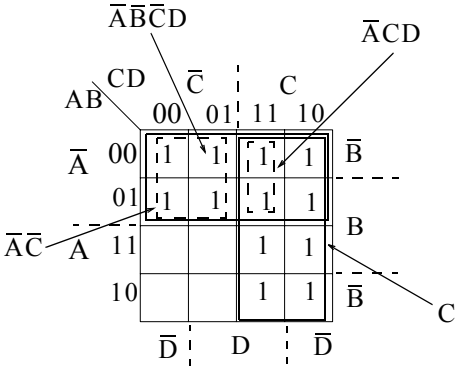


Figure 7.37. K-map for Example 7.16

The simplest expression is obtained by combining the squares shown in solid rectangles. Thus,

$$E = \overline{A} + C$$

7.4.5 Don't Care Conditions

Sometimes we do not care what value is assumed by a Boolean expression for certain combinations of variables. For example, in BCD the combination 1010 is invalid and should not occur when designing or using BCD numbers. Quite often, however, the logic designer finds it more convenient to treat such a combination as a *don't care condition*. The letter X is often used as a don't care condition. The following example demonstrates the procedure.

Example 7.17

A Boolean expression is mapped in the K-map shown in Figure 7.38 where the Xs denote don't care conditions. Derive the simplest Boolean expression in terms of the variables A, B, C, and D.

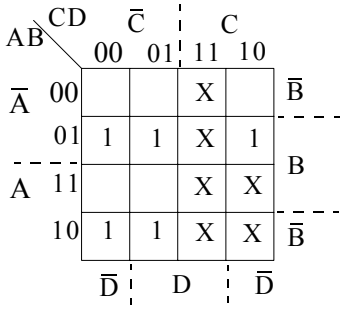


Figure 7.38. K-map for Example 7.17

Solution:

The K-map of Figure 7.38 shows two groups of 4 adjacent squares and these can be grouped as shown in Figure 7.39.

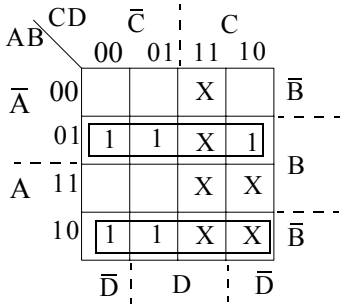


Figure 7.39. K-map for Example 7.17 with grouped variables

For this example, we interpret the Xs which would simplify our expression as 1s. Therefore, we combine the squares as shown obtaining a Boolean expression, say E, as

$$E = \bar{A}B + A\bar{B}$$

We observe that all 1s have been covered, and so we disregard the remaining Xs since they neither reduce the number of the terms nor reduce the number of variables in the expression for E.

7.5 Design of Common Logic Circuits

The following procedures are essential in designing logic circuits from the specifications or problem statements.

1. The problem must be clearly stated.
2. A truth table must be prepared from the problem description.
3. A K-map can be used to simplify the logic expression.
4. A logic diagram can be drawn from the simplest logic expression.

The examples that follow will illustrate these procedures.

7.5.1 Parity Generators/Checkers

The following example illustrates the design steps for parity generators.

Example 7.18

Design a combinational logic circuit to generate odd parity for the BCD code, i.e., a logic circuit that will generate the odd parity bit for any 4-bit combination of the BCD code.

Solution:

From the problem statement we conclude that the logic circuit we are asked to design will have 4 inputs, say A, B C, and D, where these inputs represent the weights of the BCD code (A = msb and D = lsd), and one output P representing the parity bit. This can be represented by the block diagram of Figure 7.40.

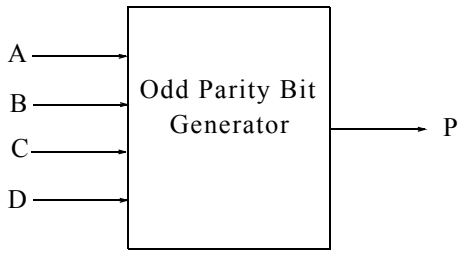


Figure 7.40. 4-bit odd parity generator for Example 7.18

Next, we construct the truth table shown as Table 7.1, and since the problem statement made no mention of the illegal BCD (1010, 1011, 1100, 1101, 1110, and 1111) combinations we treat these as don't cares.

The next step of the design process is to construct a K-map so that we can obtain the simplest Boolean expression from it. The K-map is shown in Figure 7.41.

TABLE 7.1 4-bit odd parity generator

Inputs				Output
A	**B**	**C**	**D**	**P**
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

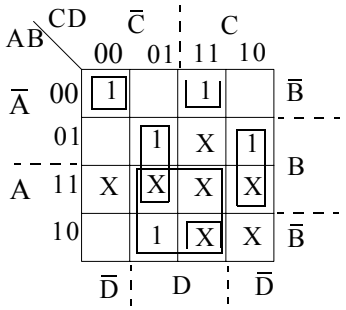


Figure 7.41. K-map for Example 7.18

The K-map of Figure 7.41 indicates that the simplest Boolean expression is

$$P = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{B}CD + B\bar{C}D + BC\bar{D} + AD$$

The logic circuit can be implemented as shown in Figure 7.42.

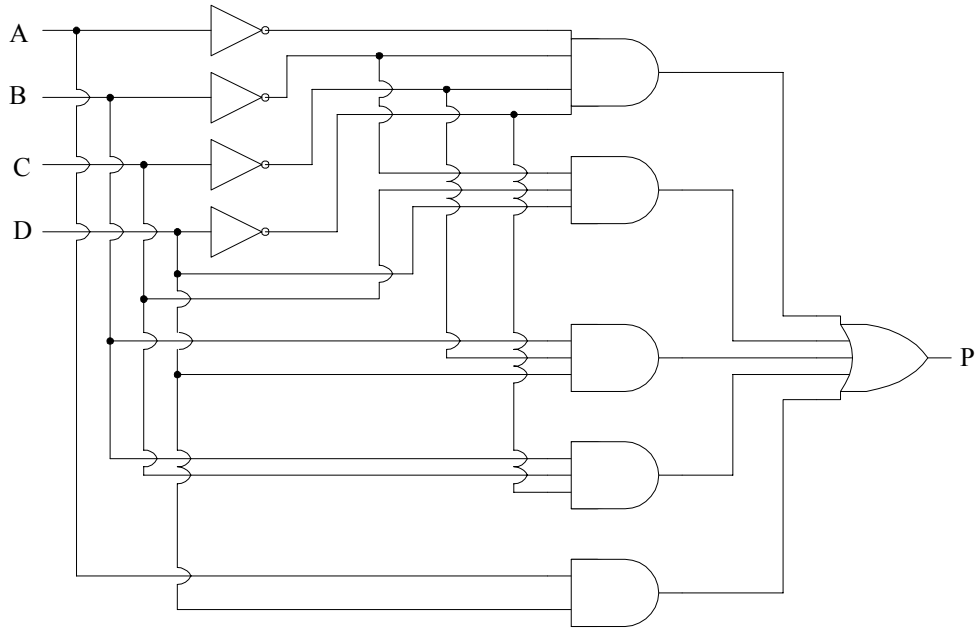


Figure 7.42. 4-bit odd parity generator for Example 7.18

Figure 7.43 shows Fairchild's 74F280 9-bit Generator/Checker. This IC is a high-speed parity generator/checker that accepts nine bits of input data (I_0 through I_8) and detects whether an even or odd number of these inputs is HIGH. If an even number of inputs is HIGH the Sum Even (Σ_E) output is HIGH. If an odd number of inputs is HIGH the Sum Even (Σ_E) output is LOW. The Sum Odd (Σ_O) output is the complement of the Sum Even (Σ_E) output. The truth table for this device is as shown in Table 7.2.

TABLE 7.2 Truth Table for Fairchild's 74F280 9-bit Generator/Checker

Number of HIGH inputs ($I_0 - I_8$)	Outputs	
	Sum Even (Σ_E)	Sum Odd (Σ_O)
0, 2, 4, 6, 8	H	L
1, 3, 5, 7, 9	L	H

7.5.2 Digital Encoders

A digital encoder circuit is a combinational circuit that accepts m input lines, one for each element of information, and generates a binary code of n output lines. A digital encoder has 2^n or less inputs and n outputs.

A block diagram of a 16-to-4 encoder is shown in Figure 7.43.

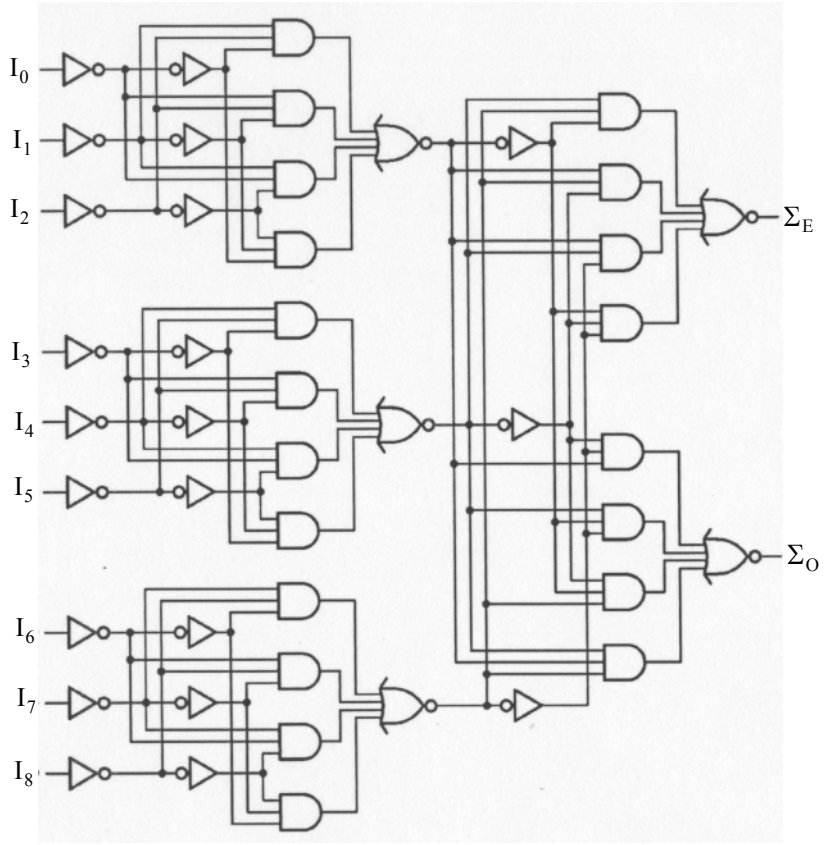


Figure 7.43. The 74F280 9-bit parity generator/checker (Courtesy Fairchild Semiconductor)

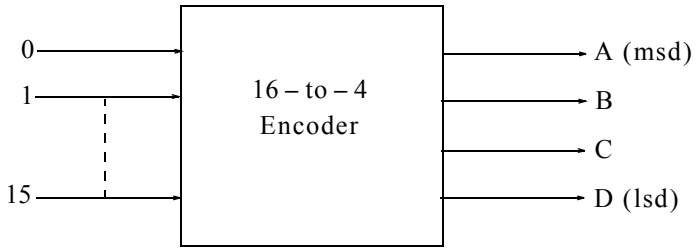


Figure 7.44. Block diagram of a 16-to-4 encoder

Example 7.19

Design an octal-to-binary encoder.

Solution:

As we know, the 8 possible combinations of groups of 3 binary numbers can be represented by octal numbers 0 through 7. Accordingly, the octal-to-binary encoder will have 8 input lines and 3 output lines as shown in Figure 7.45.

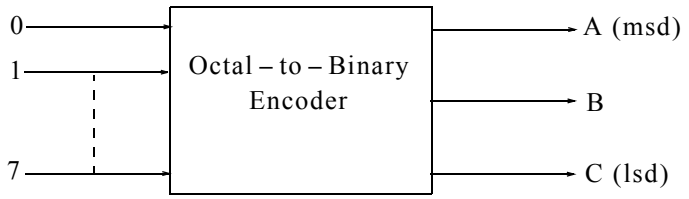


Figure 7.45. Block diagram of octal-to-binary encoder

The truth table of the octal-to-binary encoder is shown in Table 7.3 where, for convenience, the input lines are shown as D_0 through D_7.

TABLE 7.3 Truth table for octal-to-binary encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	A	B	C
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

From Table 7.3,

$$A = D_4 + D_5 + D_6 + D_7$$
$$B = D_2 + D_3 + D_6 + D_7$$
$$C = D_1 + D_3 + D_5 + D_7$$

The octal-to-binary encoder can be implemented with three 4-input OR gates as shown in Figure 7.46.

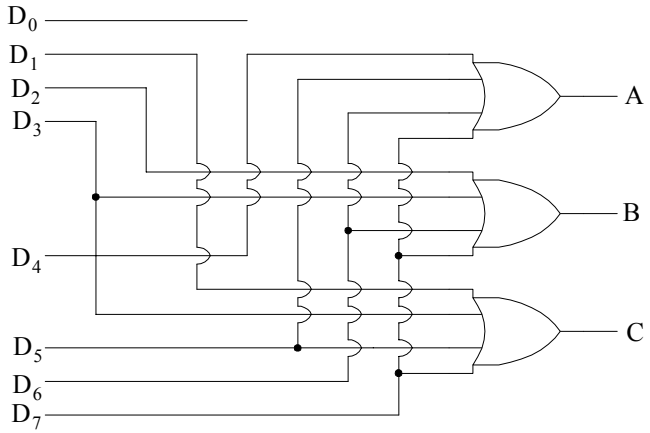


Figure 7.46. Logic diagram for octal-to-binary encoder

7.5.3 Decimal-to-BCD Encoder

The following example illustrates the design steps for a decimal-to-BCD encoder.

Example 7.20

Design a decimal-to-BCD encoder. Assume that each decimal digit (0 through 9) is represented by a switch which when closed is interpreted as logic 1 and when open is interpreted as logic 0.

Solution:

The problem statement instructs us to design a logic circuit whose inputs are the decimal numbers 0 through 9 denoted as switches S_0 through S_9 and the output is the BCD code, that is the logic circuit has ten input lines and four output lines as shown in Figure 7.47.

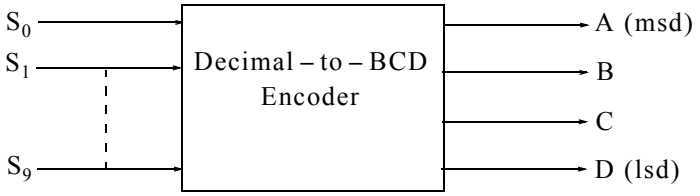


Figure 7.47. Block diagram for decimal-to-BCD converter

Obviously, only one of the ten switches S_0 through S_9 will be closed (logical 1) at any time and thus the truth table is as shown in Table 7.4.

TABLE 7.4 Truth table for decimal-to-BCD encoder

Inputs										Outputs			
S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	A	B	C	D
1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0	0	0	0	1	1	0
0	0	0	0	0	0	0	1	0	0	0	1	1	1
0	0	0	0	0	0	0	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	1	0	0	1

From the truth table above, by inspection, we get

$$A = S_8 + S_9$$
$$B = S_4 + S_5 + S_6 + S_7$$
$$C = S_2 + S_3 + S_6 + S_7$$
$$D = S_1 + S_3 + S_5 + S_7 + S_9$$

We can implement the logic circuit of the decimal-to-BCD converter with three quad 2-input OR gates (SN74LS32 IC devices) connected as shown in Figure 7.48 where D is the least significant bit and A is the most significant bit. The input S_0 is shown unconnected since none of the outputs will be logical 1. Alternately, we may say that $S_0 = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$.

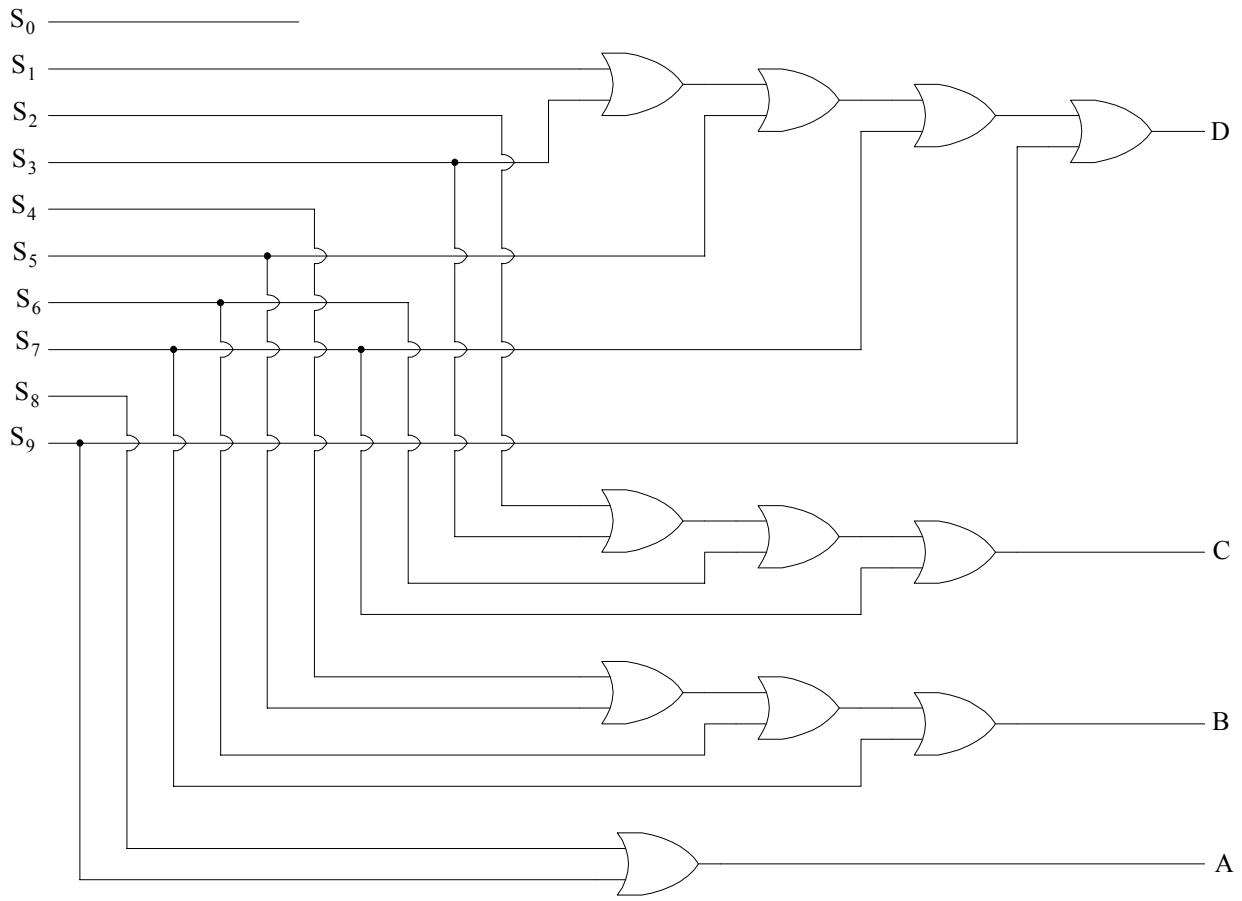


Figure 7.48. Logic diagram for decimal-to-BCD encoder

Figure 7.49 shows is Motorola's MC74HC147 decimal-to-BCD encoder logic diagram and Figure 7.50 shows the pin assignment. This device is compatible with standard CMOS outputs; with pull-up resistors are compatible with TTL devices. It converts nine active-low data inputs to four active-low BCD outputs and ensures that only the highest order active data line is converted.

The implied decimal zero condition is encoded when all nine input lines are HIGH, that is, inactive.

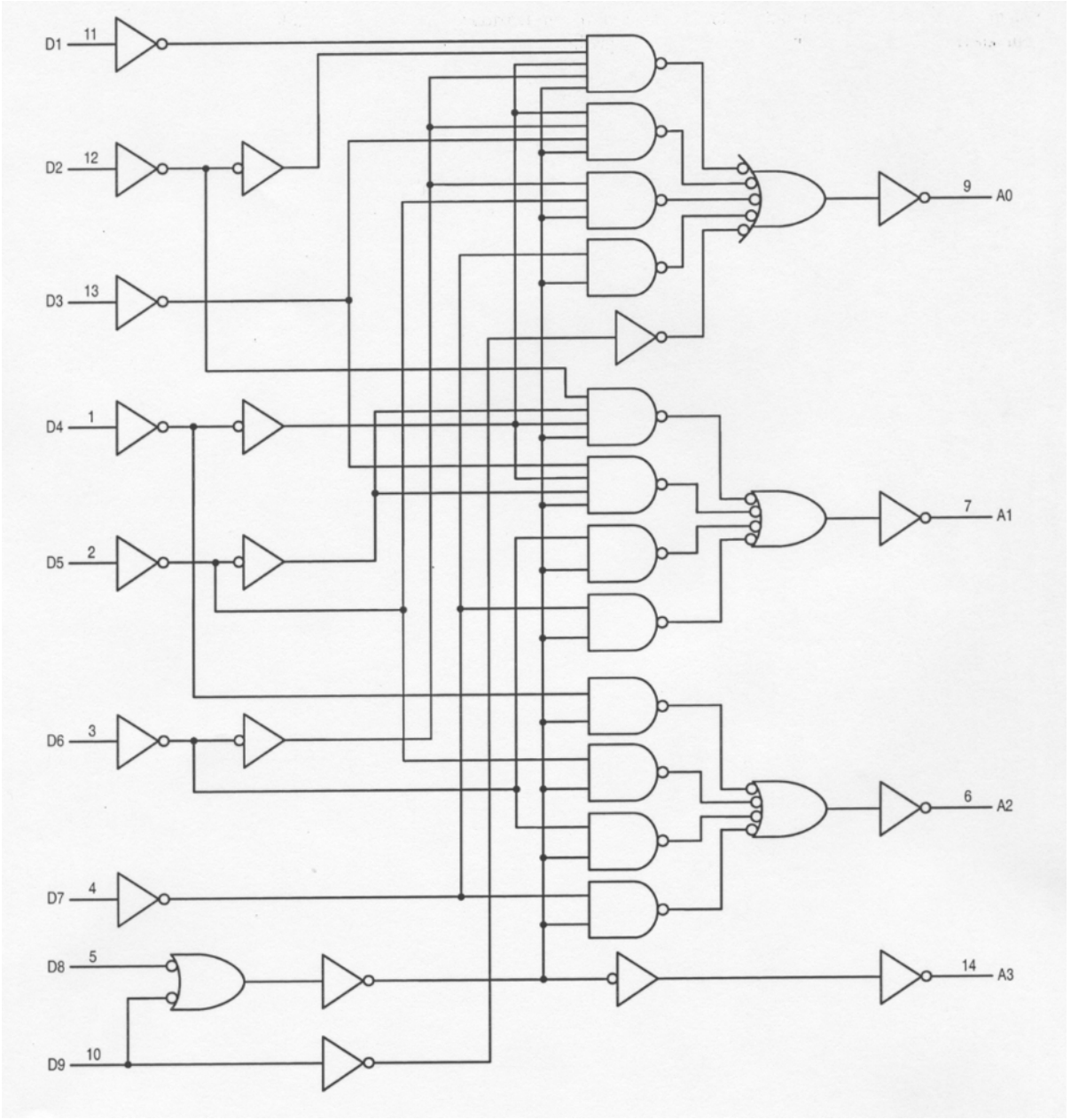


Figure 7.49. Motorola's MC74HC147 decimal-to-BCD encoder logic diagram

7.5.4 Digital Decoders

A digital decoder circuit is a combinational circuit that converts a binary code of n input lines into 2^n output lines, one for each element of information. A digital decoder has n input lines and

2^n or less outputs.

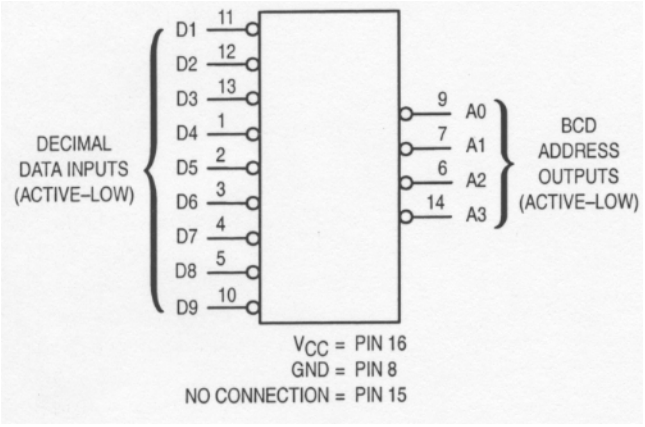


Figure 7.50. Motorola's MC74HC147 decimal-to-BCD converter pin assignment

A block diagram of a 4-to-16 decoder is shown in Figure 7.51.

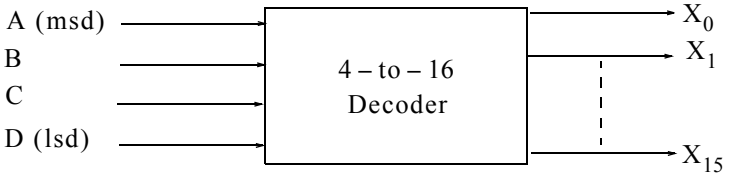


Figure 7.51. Block diagram of a 4-to-16 encoder

Example 7.21

Design a binary-to-octal decoder.

Solution:

The block diagram of a binary-to-octal decoder is shown in Figure 7.52 where the three inputs A, B, and C represent a binary number of 3 bits and the 8 outputs are the octal numbers 0 through 7.

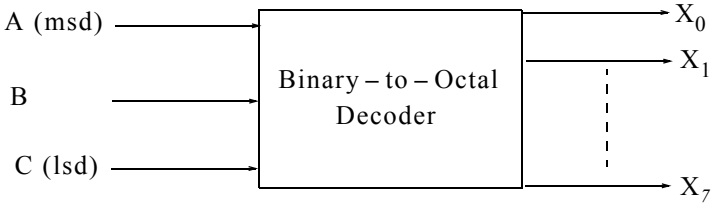


Figure 7.52. Block diagram of binary-to-octal decoder

The truth table of the octal-to-binary decoder is shown in Table 7.5 where, for convenience, the input lines are shown as X_0 through X_7.

TABLE 7.5 Truth table for binary-to-octal decoder

Inputs			Outputs							
A	B	C	X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

From Table 7.5,

$$X_0 = \overline{A}\,\overline{B}\,\overline{C}$$
$$X_1 = \overline{A}\,\overline{B}C$$
$$X_2 = \overline{A}B\overline{C}$$
$$X_3 = \overline{A}BC$$
$$X_4 = A\overline{B}\,\overline{C}$$
$$X_5 = A\overline{B}C$$
$$X_6 = AB\overline{C}$$
$$X_7 = ABC$$

The binary-to-octal decoder can be implemented with three inverters and eight 3-input AND gates as shown in Figure 7.53.

Figures 7.54, 7.55, and 7.56 show the truth table, bock diagram, and logic diagram respectively of the MC14028B binary-to-octal decoder, and BCD-to-decimal decoder combined in one IC. This device is designed so that a BCD code of 4 inputs provided a decimal (one of ten) decoded output, while the 3-bit binary input provides a decoded (one of eight) code output. It provides Low outputs on all illegal input combinations. This device is useful for code conversion, address decoding, memory selection control, demultiplexing (to be discussed later in this chapter), and readout decoding.

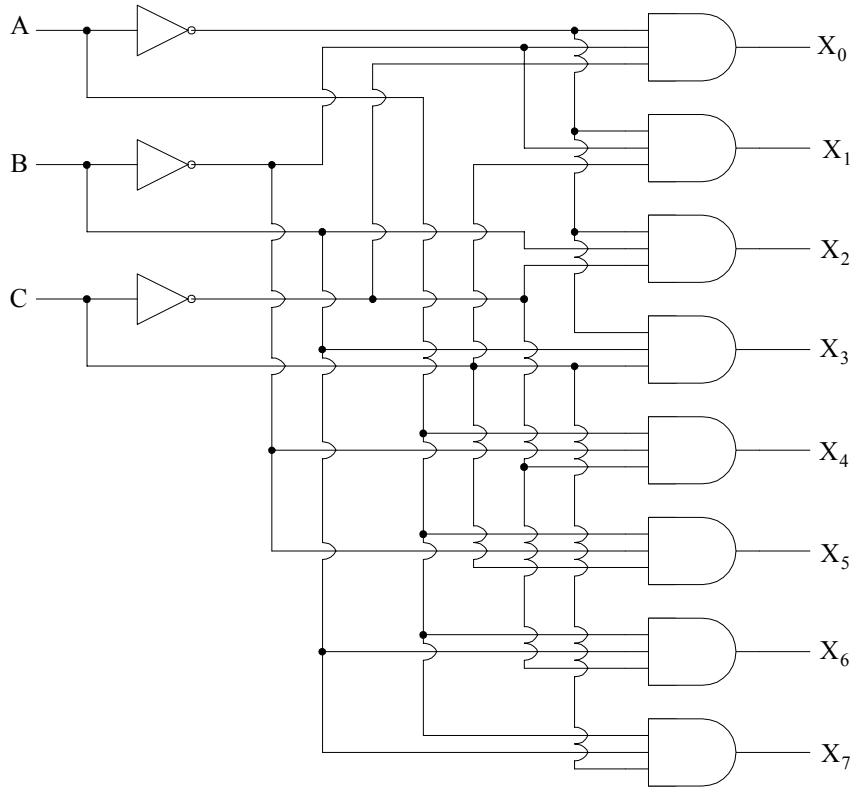


Figure 7.53. Logic diagram for the binary-to-octal decoder of Example 7.21

TRUTH TABLE

D	C	B	A	Q9	Q8	Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0
0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	1	0	0	0	0	0	0
0	1	1	1	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0

Figure 7.54. Truth table for the binary-to-octal decoder and BCD-to decimal decoder of the MC14028B (Courtesy ON Semiconductor)

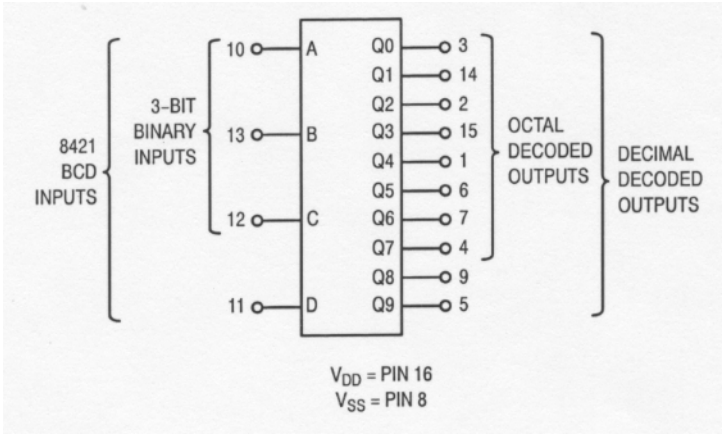


Figure 7.55. Block diagram for the binary-to-octal decoder and BCD-to decimal decoder of the MC14028B (Courtesy ON Semiconductor)

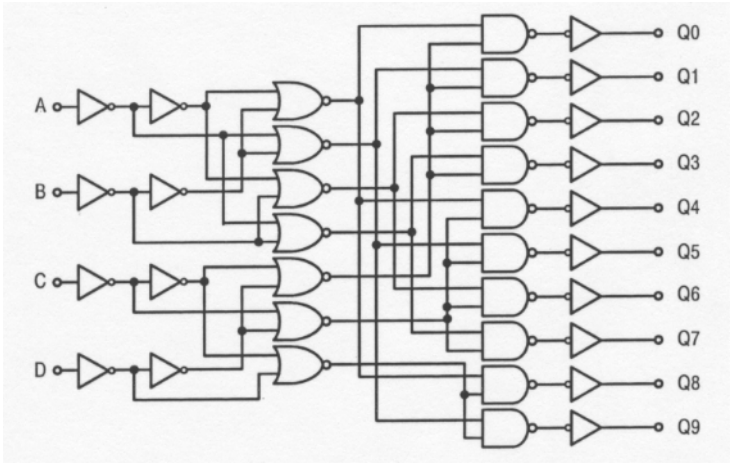


Figure 7.56. Logic diagram for the binary-to-octal decoder and BCD-to decimal decoder of the MC14028B (Courtesy ON Semiconductor)

7.5.5 Equality Comparators

An *equality comparator* is a logic circuit that produces a logical 1 output when its inputs are equal, and a logical 0 output when its inputs are unequal.

Example 7.22

Design an equality comparator that compares two 2-bit words (A, B) where each word has two parallel inputs $A_0 - A_1$, $B_0 - B_1$, and A_1, B_1 are the most significant bits.

Solution:

The block diagram of a two 2-bit words equality comparator is shown in Figure 7.57, and the

four-input combinations which will make the comparator output X logical 1 are those for which $A_1A_0 = B_1B_0$.

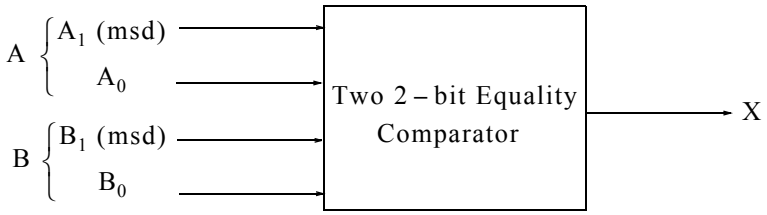


Figure 7.57. Block diagram two 2-bit equality comparator

The truth table is shown in Table 7.6 and the K-map in Figure 7.58.

TABLE 7.6 Truth table for two 2-bit equality comparator

Inputs				Output
A_1	A_0	B_1	B_0	X
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

The K-map of Figure 7.58 shows that the logical 1s map in a diagonal and thus

$$X = \overline{A_1}\,\overline{A_0}\,\overline{B_1}\,\overline{B_0} + \overline{A_1}A_0\overline{B_1}B_0 + A_1\overline{A_0}B_1\overline{B_0} + A_1A_0B_1B_0$$

The same result will be obtained using a different but more simplified form if we combine the 0s in the K-map as shown in Figure 7.59.

The combinations of 0s will produce a Boolean expression in complemented form, for this example the output X will be complemented. Thus, the K-map of Figure 7.59 yields the logical expression

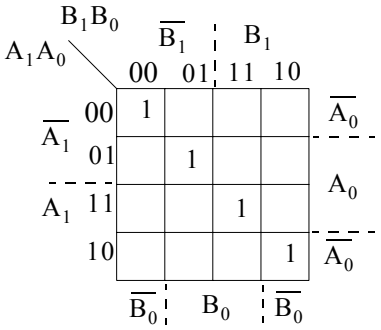


Figure 7.58. K-map for Example 7.22

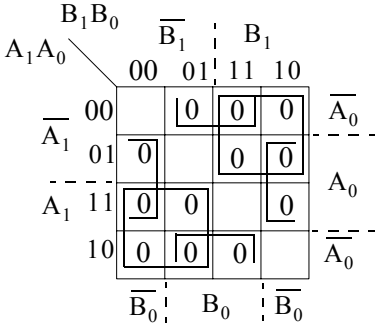


Figure 7.59. Alternate K-map form for Example 7.22

$$\overline{X} = A_1\overline{B_1} + \overline{A_1}B_1 + A_0\overline{B_0} + \overline{A_0}B_0 = A_1 \oplus B_1 + A_0 \oplus B_0$$

and the output X in uncomplemented form is

$$X = \overline{A_1 \oplus B_1 + A_0 \oplus B_0}$$

The last expression above can be implemented with two 2-input XOR gates followed by a 2-input NOR gate as shown in Figure 7.60.

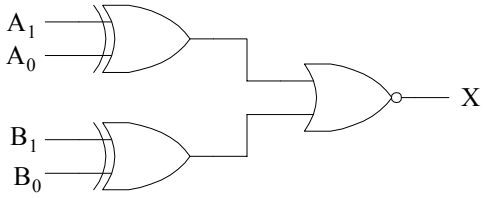


Figure 7.60. Logic diagram for a two 2-bit equality comparator

The SN74LS85 IC device is a 4-bit magnitude comparator and compares two 4-bit words (A, B)

where each word has four parallel inputs $A_0 - A_3$, $B_0 - B_3$, and A_3, B_3 are the most significant bits. Figures 7.61 and 7.62 show the logic diagram and the truth table respectively for this device.

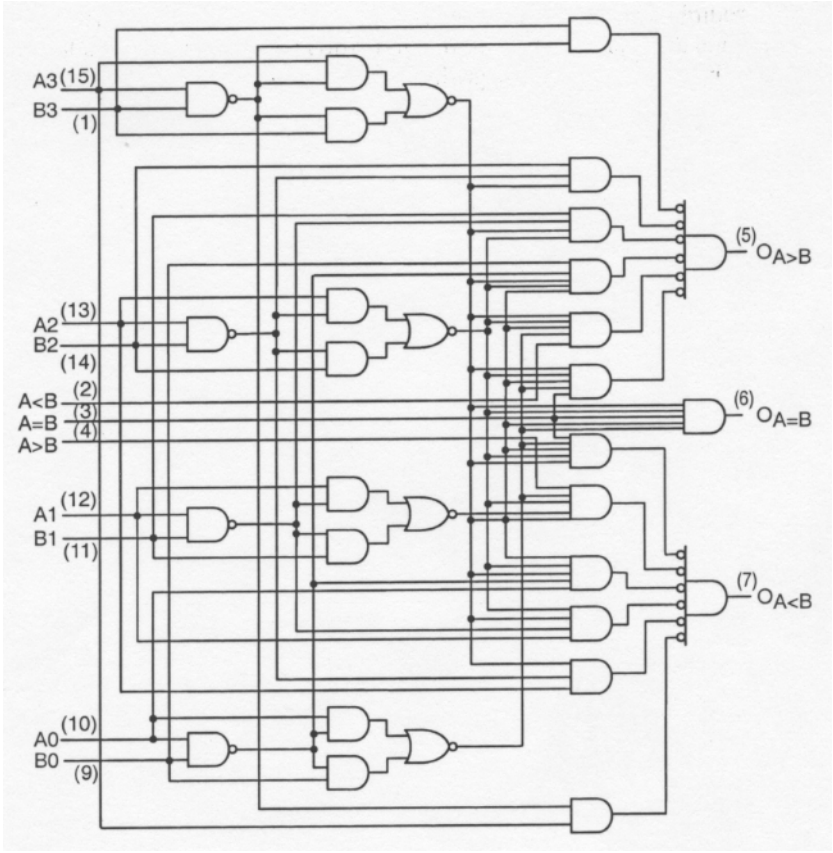


Figure 7.61. Logic diagram for the SN74LS85 4-bit magnitude comparator (Courtesy Texas Instruments)

TRUTH TABLE

COMPARING INPUTS				CASCADING INPUTS			OUTPUTS		
A_3,B_3	A_2,B_2	A_1,B_1	A_0,B_0	$I_{A>B}$	$I_{A<B}$	$I_{A=B}$	$O_{A>B}$	$O_{A<B}$	$O_{A=B}$
$A_3>B_3$	X	X	X	X	X	X	H	L	L
$A_3<B_3$	X	X	X	X	X	X	L	H	L
$A_3=B_3$	$A_2>B_2$	X	X	X	X	X	H	L	L
$A_3=B_3$	$A_2<B_2$	X	X	X	X	X	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1>B_1$	X	X	X	X	H	L	L
$A_3=B_3$	$A_2=B_2$	$A_1<B_1$	X	X	X	X	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1=B1$	$A_0>B_0$	X	X	X	H	L	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0<B_0$	X	X	X	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	H	L	L	H	L	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	L	H	L	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	X	X	H	L	L	H
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	H	H	L	L	L	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	L	L	L	H	H	L

H = HIGH Level
L = LOW Level
X = IMMATERIAL

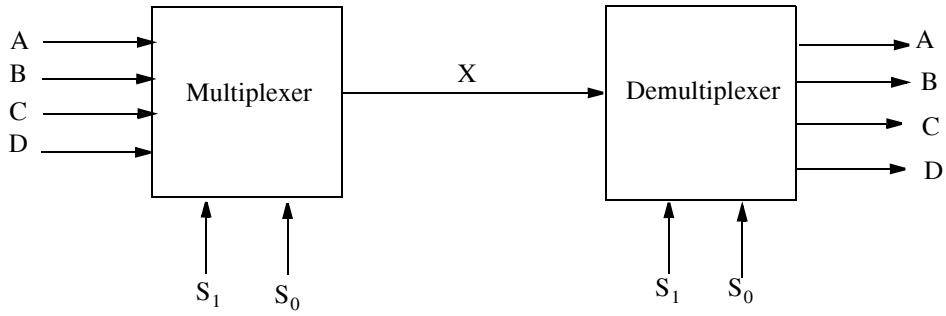
Figure 7.62. Truth table for the SN74LS85 4-bit magnitude comparator

7.5.6 Multiplexers and Demultiplexers

Multiplexing is a method of sending multiple signals or streams of information on a carrier at the same time in the form of a single, complex signal and then recovering the separate signals at the receiving end. Analog signals are commonly multiplexed using *Frequency Division Multiplexing* (FDM), in which the carrier bandwidth is divided into subchannels of different frequency widths, each carrying a signal at the same time in parallel. Cable television is an example of FDM.

Digital signals are commonly multiplexed using *Time Division Multiplexing* (TDM), in which the multiple signals are carried over the same channel in alternating time slots.

A *demultiplexer* performs the reverse operation of a multiplexer. Figure 7.63 shows a functional block diagram of a typical 4-line time-division multiplexer / demultiplexer.



Truth Table

S_1	S_0	X
0	0	A
0	1	B
1	0	C
1	1	D

Figure 7.63. Functional Block Diagram for 4-line time-division multiplexer/demultiplexer

In Figure 7.63, *A, B, C,* and *D* represent input data to be multiplexed and appear on a single transmission path denoted as *X*. This path will carry the data of input *A* or *B* or C or *D* depending on the settings of the selection switches S_0 and S_1. These setting must be the same on both the multiplexer and demultiplexer. For instance, if the setting are $S_0 = 0$ and $S_1 = 1$, the output line *X* of the multiplexer will carry the data of signal C and it will appear at the output line C on the demultiplexer. The other combinations are shown in the truth table of Figure 7.63.

Example 7.23

Design an 8-input digital multiplexer.

Solution:

The block diagram for this digital multiplexer is shown in Figure 7.64 and the truth table is shown in Table 7.7.

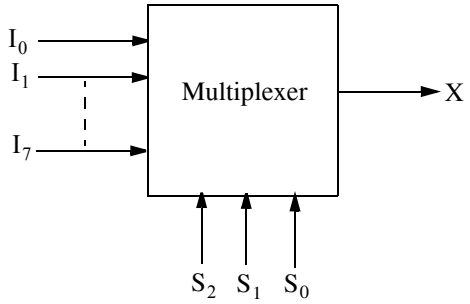


Figure 7.64. 8-input digital multiplexer for Example 7.23

TABLE 7.7 Truth table for 8-input digital multiplexer

Control lines			Outputs							
S_2	S_1	S_0	$X = I_0$	$X = I_1$	$X = I_2$	$X = I_3$	$X = I_4$	$X = I_5$	$X = I_6$	$X = I_7$
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

From the truth table of Table 7.7 we obtain the Boolean expression

$$X = I_0\overline{S}_0\overline{S}_1\overline{S}_2 + I_1S_0\overline{S}_1\overline{S}_2 + I_2\overline{S}_0S_1\overline{S}_2 + I_3S_0S_1\overline{S}_2 + I_4\overline{S}_0\overline{S}_1S_2 + I_5S_0\overline{S}_1S_2 + I_6\overline{S}_0S_1S_2 + I_7S_0S_1S_2$$

The logic diagram for the 8-input multiplexer is shown in Figure 7.65.

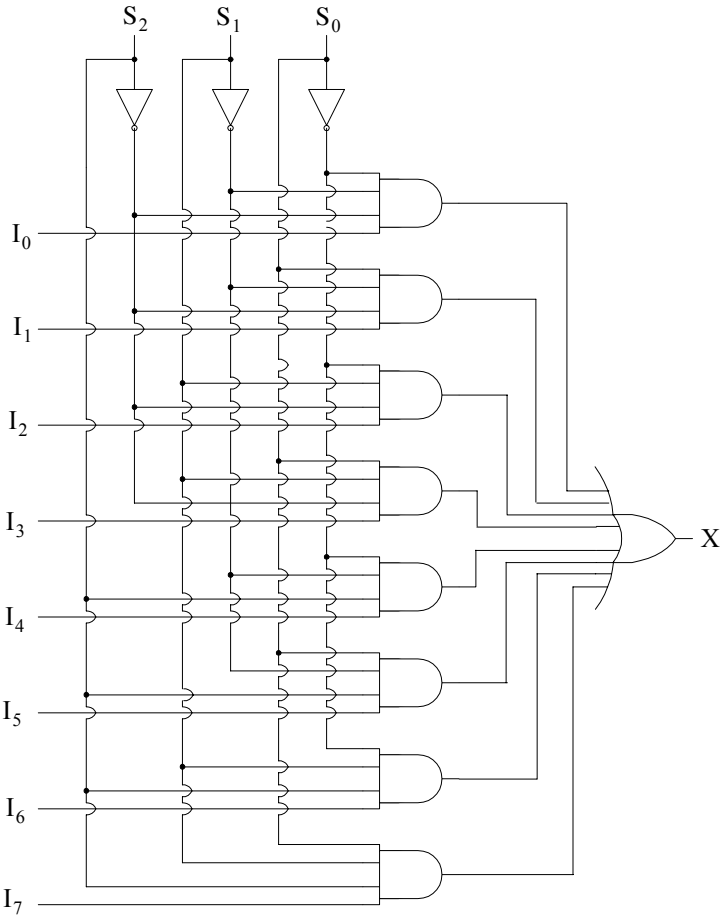


Figure 7.65. Logic diagram of 8-input digital multiplexer for Example 7.23

A commercially available multiplexer is the SN74LS251 8-to-1 line 3-state data selector/multiplexer with complementary outputs.

Example 7.24

Design an 8-output digital demultiplexer.

Solution:

The digital demultiplexer performs the reverse operation of the multiplexer, that is, a single input line is directed to any of the outputs under control of the selection lines. The block diagram for an 8-output digital demultiplexer is shown in Figure 7.66 and the truth table is shown in Table 7.8.

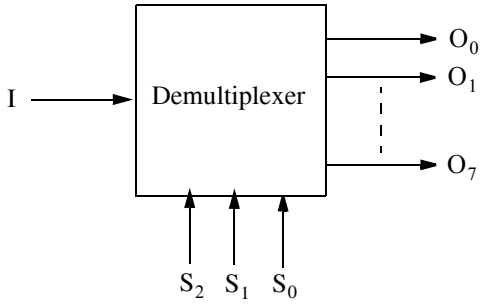


Figure 7.66. Block diagram of the 8-output digital demultiplexer for Example 7.24

TABLE 7.8 Truth table for 8-output digital demultiplexer

Control lines			Outputs							
S_2	S_1	S_0	$I \rightarrow O_0$	$I \rightarrow O_1$	$I \rightarrow O_2$	$I \rightarrow O_3$	$I \rightarrow O_4$	$I \rightarrow O_5$	$I \rightarrow O_6$	$I \rightarrow O_7$
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

From the truth table of Table 7.8 we obtain

$$O_0 = I\overline{S_0}\,\overline{S_1}\,\overline{S_2}$$
$$O_1 = I\overline{S_0}\,\overline{S_1}S_2$$
$$O_2 = I\overline{S_0}S_1\overline{S_2}$$
$$O_3 = I\overline{S_0}S_1S_2$$
$$O_4 = IS_0\overline{S_1}\,\overline{S_2}$$
$$O_5 = IS_0\overline{S_1}S_2$$
$$O_6 = IS_0S_1\overline{S_2}$$
$$O_7 = IS_0S_1S_2$$

The logic diagram for the 8-output demultiplexer is shown in Figure 7.67. A digital demultiplexer can be used as a decoder if the single input I is permanently connected to a logical 1 signal. This can be seen by comparing the logic diagrams of Figures 7.53 (binary-to-octal decoder) and 7.67 (8-output digital demultiplexer).

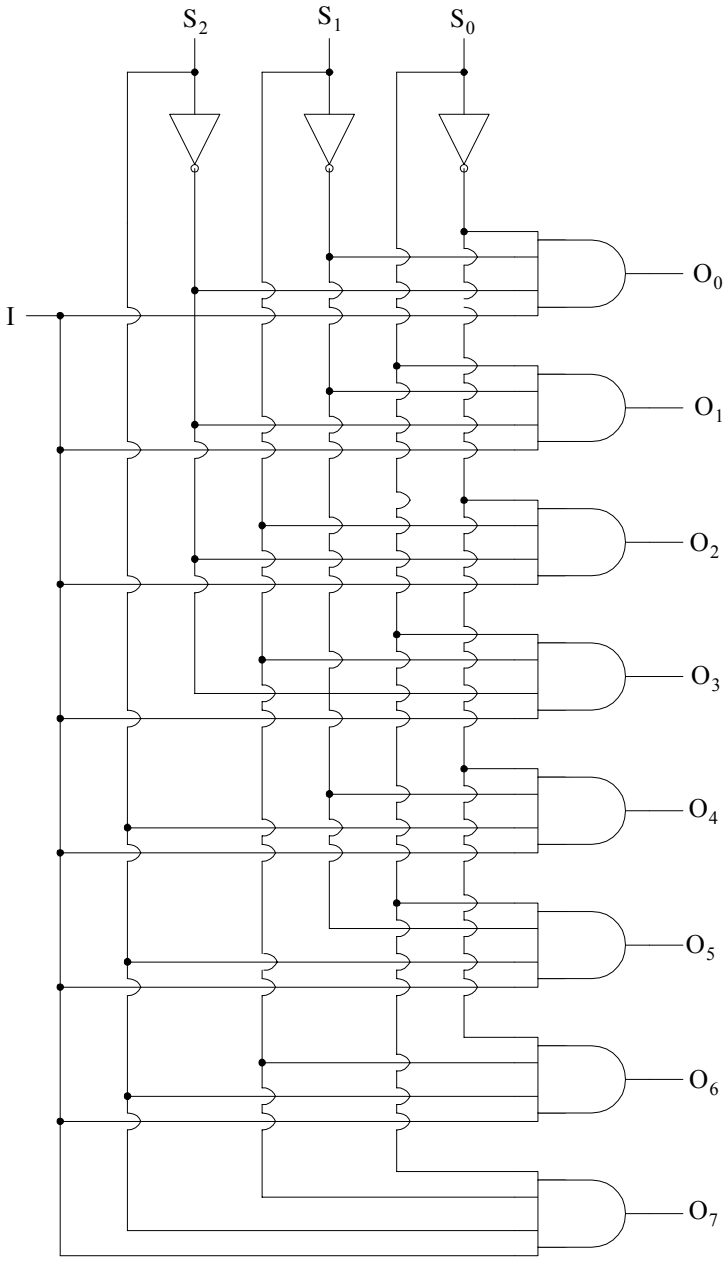


Figure 7.67. Logic diagram of the 8-output digital demultiplexer for Example 7.24

Figures 7.68 and 7.69 show the 4-to-16 line decoder/demultiplexer, device 74HC154, truth table and logic diagram respectively.

INPUT						OUTPUT														
$\overline{E0}$	$\overline{E1}$	A0	A1	A2	A3	$\overline{Y0}$	$\overline{Y1}$	$\overline{Y2}$	$\overline{Y3}$	$\overline{Y4}$	$\overline{Y6}$	$\overline{Y7}$	$\overline{Y8}$	$\overline{Y2}$	$\overline{Y9}$	$\overline{Y10}$	$\overline{Y11}$	$\overline{Y12}$	$\overline{Y13}$	$\overline{Y14}$	$\overline{Y15}$
H	H	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	L	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	H	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	L	L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
		H	L	L	L	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H
		L	H	L	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H
		H	H	L	L	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H
		L	L	H	L	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H
		H	L	H	L	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H
		L	H	H	L	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H
		H	H	H	L	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H
		L	L	L	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H
		H	L	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H
		L	H	L	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H
		H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H
		L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H
		H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H
		L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H
		H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L

Note

1. H = HIGH voltage level
   L = LOW voltage level
   X = don't care.

Figure 7.68. Truth table for 74HC154 4-to-16 decoder demultiplexer (Courtesy Philips Semiconductor)

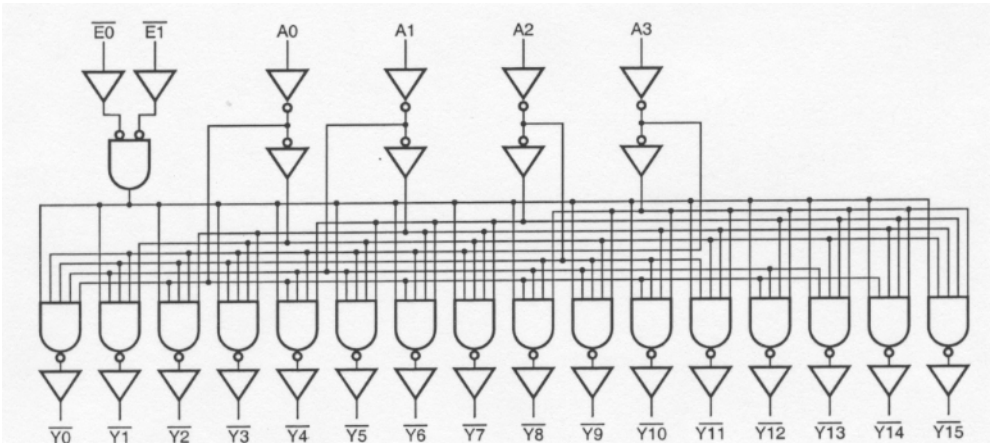


Figure 7.69. Logic diagram for 74HC154 4-to-16 decoder demultiplexer (Courtesy Philips Semiconductor)

7.5.7 Arithmetic Adder and Subtractor Logic Circuits

In this subsection we will learn how the binary arithmetic operations of addition and subtraction are performed.

Half Adder

The *half adder* is a combinational logic circuit that adds two bits. This circuit has two inputs, the augend and the addend bits, and two outputs, the sum and carry bits. The truth table shown in Table 7.9 is constructed from the *binary arithmetic operations*

$$0+0=0 \qquad 0+1=1 \qquad 1+0=1 \qquad 1+1=10 \text{ that is, Sum} = 0 \text{ and Carry} = 1$$

Let the inputs be A and B, and the outputs be S (sum) and C (carry). The truth table is shown below as Table 7.9.

TABLE 7.9 Truth table for half adder

Inputs		Outputs	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

This truth table reveals that the sum bit S is the XOR function of the inputs A and B, that is, $S = A \oplus B$, and the carry bit C is zero unless both inputs A and B are 1s. Thus, $C = AB$. The half adder logic circuit can be implemented with one XOR gate and one AND gate as shown in Figure 7.70.

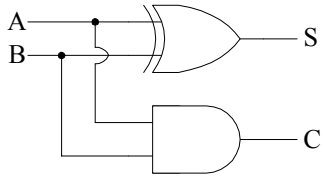


Figure 7.70. Logic circuit for the half adder

Half Subtractor

The *half subtractor* is a combinational logic circuit that subtracts one bit from another. This circuit has two inputs, the minuend and the subtrahend bits, and two outputs, the difference and borrow bits. The truth table shown in Table 7.10 is constructed from the *binary arithmetic operations*

$$0-0=0 \qquad 0-1=1 \text{ with Borrow} = 1 \qquad 1-0=1 \qquad 1-1=0$$

Let the inputs be X for the minuend and Y for the subtrahend, and the outputs be D (difference) and B (borrow). The truth table is shown below as Table 7.10.

TABLE 7.10 Truth table for half subtractor

Inputs		Outputs	
X	Y	D	B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

This truth table reveals that the difference bit D is the XOR function of the inputs X and Y, that is, $D = X \oplus Y$, and the borrow bit B is zero unless $X = 0$ and $Y = 1$. Thus, $B = \overline{X}Y$. The half subtractor logic circuit can be implemented with one XOR gate, one inverter, and one AND gate as shown in Figure 7.71.

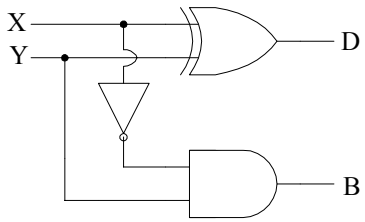


Figure 7.71. Logic circuit for the half subtractor

A comparison of the half adder with the half subtractor reveals that the outputs S (sum of the half adder) and D (difference of the half subtractor) are the same. We see also that a half adder can be converted to a half subtractor by inverting the input (minuend) X to the input of the AND gate but not to the input of the XOR gate.

Full Adder

A *full adder* circuit is a combinational circuit that produces the arithmetic addition of three input bits, where the first bit X represents the augend, the second bit Y represents the addend, and the third bit C_{IN} represents the carry from the previous lower significant bit position. The two outputs are S (sum) and C_{OUT} (carry). The truth table and logic circuit of the full adder are shown in Table 7.11 and Figure 7.74 respectively.

The K-maps for the S (sum) and C_{OUT} (carry) outputs are shown in Figures 7.72 and 7.73 respectively. The K-map of Figure 7.72 shows that no simplification for the S (sum) output is possible and thus

$$S = \overline{X}\overline{Y}C_{IN} + \overline{X}Y\overline{C}_{IN} + X\overline{Y}\overline{C}_{IN} + XYC_{IN}$$

TABLE 7.11 Truth table for full adder

Inputs			Outputs	
X	Y	C_{IN}	S	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

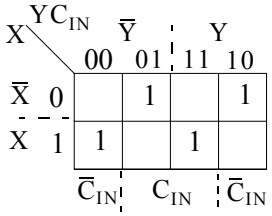


Figure 7.72. K-map for the sum bit of the full adder

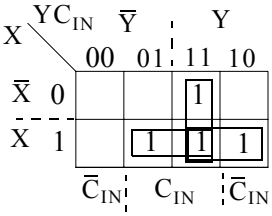


Figure 7.73. K-map for the carry bit of the full adder

The expression for the sum can also be written in terms of the XOR function as follows:

$$S = \bar{X}\bar{Y}C_{IN} + \bar{X}Y\bar{C}_{IN} + X\bar{Y}\bar{C}_{IN} + XYC_{IN} = (XY + \bar{X}\bar{Y})C_{IN} + (\bar{X}Y + X\bar{Y})\bar{C}_{IN}$$

$$= \overline{(X \oplus Y)}C_{IN} + (X \oplus Y)\bar{C}_{IN} = X \oplus Y \oplus C_{IN}$$

The K-map for the C_{OUT} (carry) indicates that some simplifications are possible, therefore,

$$C_{OUT} = XY + XC_{IN} + YC_{IN}$$

The logic diagram for the full adder is shown in Figure 7.74.

A full adder circuit can also be implemented with two half adders and one OR gate connected as shown in Figure 7.75.

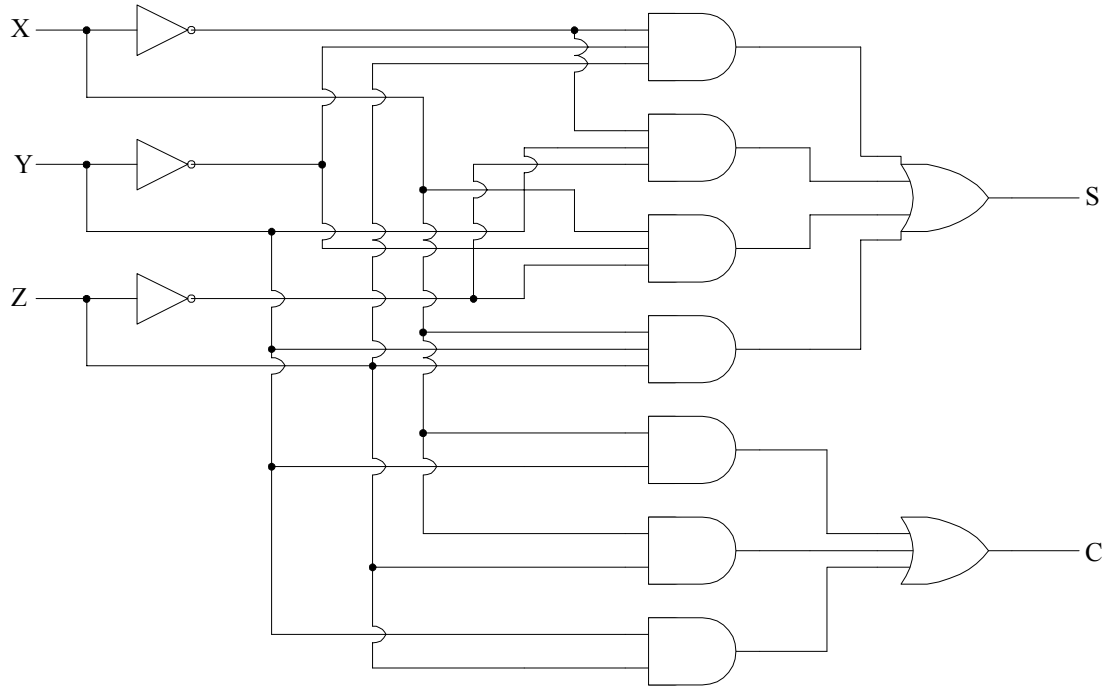


Figure 7.74. Logic diagram for the full adder

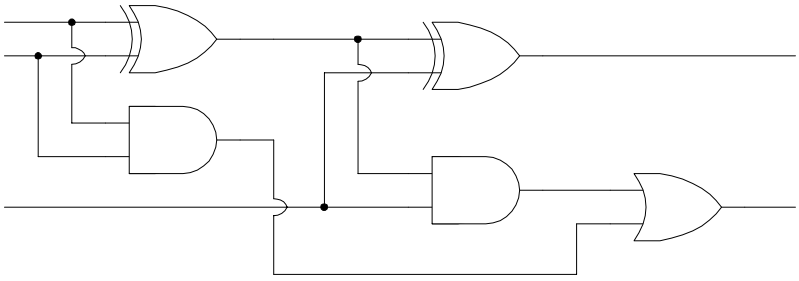


Figure 7.75. Logic diagram for a full adder with 2 half adders

Figure 7.76 shows the 74F283 4-bit binary full adder with fast carry.

Full Subtractor

A *full subtractor* is a combinational circuit that subtracts two bits and takes into account a borrow bit which may have occurred by the subtraction of the next lower position of two bits. A full subtractor has three inputs, the minuend X, the subtrahend Y, and the previous borrow Z, and two outputs, the difference D, and the present borrow B.

The truth table for the Full Subtractor is shown in Table 7.12. From this truth table we obtain the K-maps shown in Figures 7.77 and 7.78 to simplify the resulting logic expressions. The K-map for the D (difference) output indicates that no simplification is possible, therefore,

$$D = \overline{X}\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\overline{Z} + XYZ$$

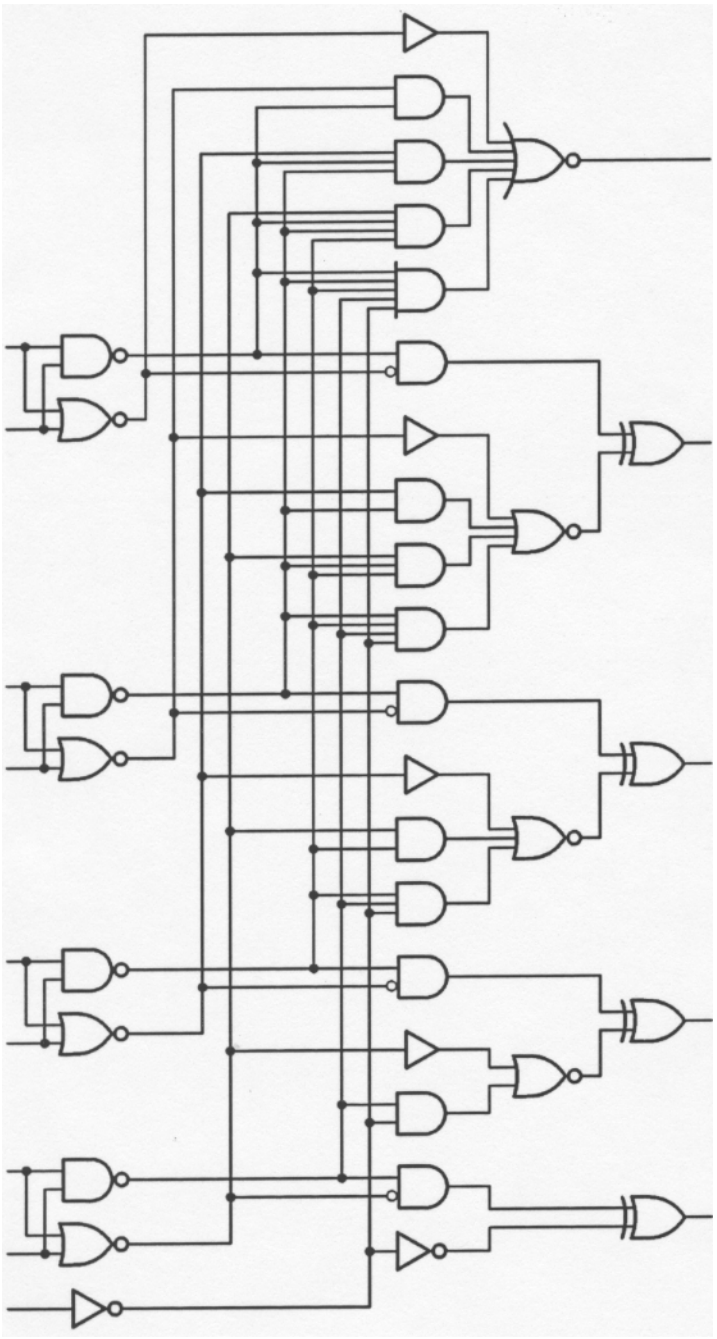


Figure 7.76. The 74F283 4-bit binary full adder with fast carry (Courtesy National Semiconductor)

The K-map for the B (borrow) output indicates that some simplifications are possible, therefore,

$$B = \overline{X}Y + \overline{X}Z + YZ$$

TABLE 7.12 Truth table for full subtractor

Inputs			Outputs	
X	Y	Z	D	B
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

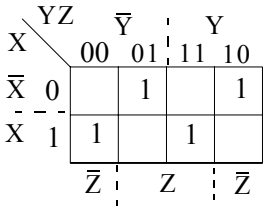


Figure 7.77. K-map for the difference bit of the full subtractor

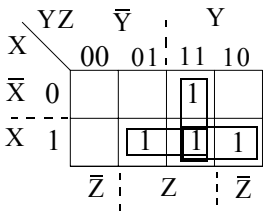


Figure 7.78. K-map for the borrow bit of the full subtractor

As before, we observe that the output D (difference) of the full subtractor is the same as the output S (sum) of the full adder circuit, and the output B (borrow) will be the same as the output C (carry) of the full adder if the minuend (input X) is complemented. Thus, it is possible to use a full adder circuit as a full subtractor circuit if we invert the minuend input X for the part of the logic circuit that produces the borrow.

Binary addition, subtraction, multiplication, and division can be easily performed with register circuits as we will see on the next chapter.

7.6 Summary

- The ANDing operation has precedence over the ORing operation. For instance, the Boolean expression $AB + C$ implies that A must first be ANDed with B and the result must be ORed with C.
- Timing diagrams are waveforms showing changing outputs with changing inputs.
- A Karnaugh map, or simply a K-map, is a matrix of squares. In general, a Boolean expression with n variables can be represented by a K-map of 2^n ì squares where each square represents a row of an equivalent truth table. A K-map provides a very powerful method of reducing Boolean expressions to their simplest forms. To use a K-map effectively, we must follow the procedures below.

  1. Each square with a 1 in it must be included at least once in the combinations of the squares which were selected. However, a particular square may be included in more than one combination of squares if it can be combined with other squares.
  2. The number of squares to be combined must be a power of 2, i.e., squares can be combined in groups of two, four, eight, and so on squares.
  3. Each combination should be selected to include the highest possible number of squares.

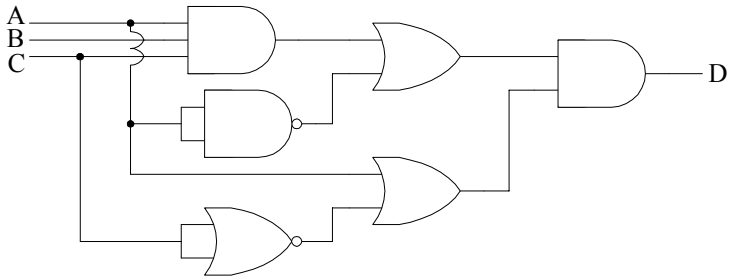
- Quite often, we do not care if a variable represents a logical 1 or a logical 0. In this case it is convenient to represent that variable as a don't care usually with the letter X. Don't care conditions allow us to obtain simpler Boolean expressions.
- Parity generators are logic circuits where a bit is added to implement odd or even parity.
- A digital encoder circuit is a combinational circuit that accepts m input lines, one for each element of information, and generates a binary code of n output lines. A digital encoder has 2^n or less inputs and n outputs.
- A digital decoder circuit is a combinational circuit that converts a binary code of n input lines into 2^n output lines, one for each element of information. A digital decoder has n input lines and 2^n or less outputs.
- An equality comparator is a logic circuit that produces a logical 1 output when its inputs are equal, and a logical 0 output when its inputs are unequal.
- A digital multiplexer is a logic circuit with 2^n input lines and only one output. The output can be any of the inputs by proper selection of control switches.
- A demultiplexer performs the reverse operation of a multiplexer. It has one input line and 2^n output lines.

- The half adder is a combinational logic circuit that adds two bits. This circuit has two inputs, the augend and the addend bits, and two outputs, the sum and carry bits.
- The half subtractor is a combinational logic circuit that subtracts one bit from another. This circuit has two inputs, the minuend and the subtrahend bits, and two outputs, the difference and borrow bits.
- A full adder circuit is a combinational circuit that produces the arithmetic addition of three input bits, where the first bit X represents the augend, the second bit Y represents the addend, and the third bit Z represents the carry from the previous lower significant bit position. The two outputs are S (sum) and C (carry).
- A full subtractor is a combinational circuit that subtracts two bits and takes into account a borrow bit which may have occurred by the subtraction of the next lower position of two bits. A full subtractor has three inputs, the minuend X, the subtrahend Y, and the previous borrow Z, and two outputs, the difference D, and the present borrow B.

7.7 Exercises

For all exercises, assume that the variables are available in the uncomplemented form only.

1. Implement the theorem $A + \overline{A} = 1$ using
   a. NAND gates only
   b. NOR gates only
2. For the logic diagram below, derive the Boolean expression and simplify it if possible. Then, implement the logic diagram for the simplified expression.



3. Use a single SN74LS00 Quad 2-input NAND gate IC device to implement the Boolean expression $E = \overline{(\overline{A} + \overline{B})}CD$.
4. Use NAND gates only to implement the Boolean expression $W = X\overline{Y} + Y\overline{Z} + \overline{X}Z$.
5. For this exercise assume that logical 0 = 0 volts = Low, logical 1 = 5 volts = High, and the inputs to AND, NAND, OR, NOR, XOR, and XNOR gates are A = 11000101 and B = 10110010 during the time interval $T_1 \le T \le T_2$. Sketch the timing diagrams for AB, $\overline{AB}$, $A + B$, $\overline{A + B}$, $A \oplus B$, and $\overline{A \oplus B}$ for this time interval.
6. A Boolean expression is mapped in the K-map shown below. Derive the simplest Boolean expression in terms of the variables A, B, C, and D.

AB \ CD	00	01	11	10
00		1		
01		1	1	1
11	1	1	1	
10			1	

7. A Boolean expression is mapped in the K-map shown below. Derive the simplest Boolean expression in terms of the variables A, B, C, and D.

AB \ CD	00	01	11	10
00	1		1	
01	1	1	1	1
11	1		1	1
10		1		

For Exercises 8 through 14, derive the truth tables and by the use of K-maps derive the simplest Boolean expressions for the output(s). There is no need to draw the logic diagrams.

8. Design a two 2-bit input equality / inequality comparator logic circuit with three outputs, the first for the condition $A < B$, the second for the condition $A = B$, and the third for the condition $A > B$.

9. Design a 4-to-2 encoder logic circuit.

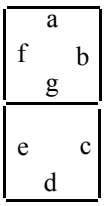
10. Design a 2-to-4 decoder logic circuit.

11. Design a 4-input digital multiplexer circuit.

12. Design a 4-output digital demultiplexer circuit.

13. Change the logic circuit of the full adder so that it will perform as a full subtractor.

14. *Seven segment displays* are ICs consisting of *light emitting diodes* (LEDs). The seven segments are arranged as shown below. Digital clocks, counters, and other equipment use seven segment displays.



By lighting certain combinations of these seven segments (a through g), we can display any decimal character 0 through 9. For example, when segments a, b, and c are lit, the number 7 is displayed. Likewise, segments b and c will display the number 1.

A seven segment display operates in conjunction with another IC called *seven segment decoder/driver* that contains the necessary logic to decode a 4-bit BCD code in order to drive the segments of the seven segment display IC.

a. Design a seven segment decoder/driver that will decode the 4-bit BCD code to the equiva-

lent decimal characters 0 through 9. The unused combinations 1010. 1011, 1100, 1101. 1110. and 1111 can be considered as don't cares. Derive the simplest Boolean expressions for each of the seven segments (a through g).

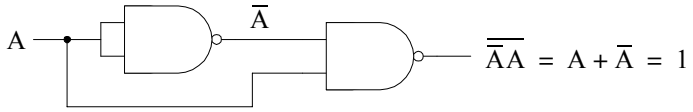
b. Show the seven segment displays when the inputs are the unused combinations 1010. 1011, 1100. 1101. 1110. and 1111.

c. Redesign the seven segment decoder/driver so that the unused combinations will produce the hexadecimal outputs A, b, C, d, E, and F corresponding to the inputs are 1010, 1011, 1100, 1101, 1110, and 1111 respectively. Lower case for the letters B and D are chosen so they will not look like 8 and 0.

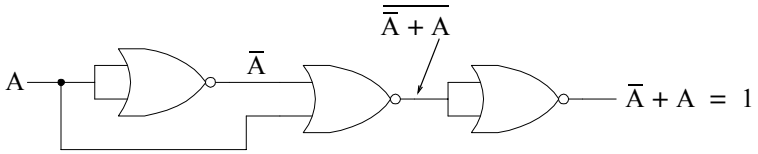
7.8 Solutions to End-of-Chapter Exercises

1.

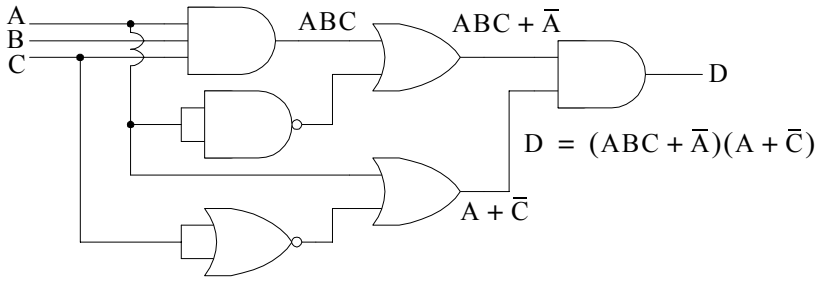
a.



b.



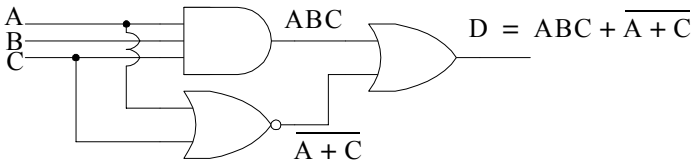
2.



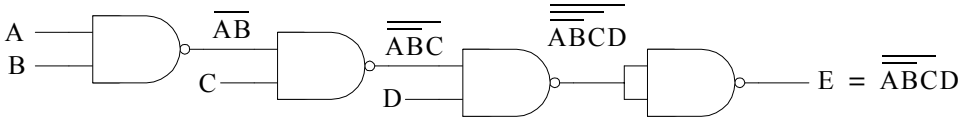
Simplifying the expression for D we get

$$D = (ABC + \overline{A})(A + \overline{C}) = AABC + ABC\overline{C} + A\overline{A} + \overline{A}\,\overline{C}$$
$$= ABC + 0 + 0 + \overline{A}\,\overline{C} = ABC + \overline{A + C}$$

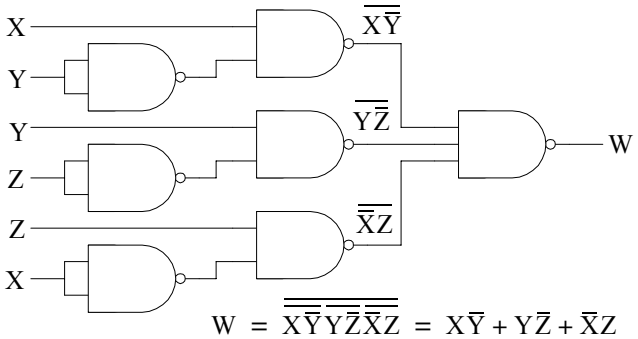
The last expression is implemented with the logic diagram below.



3.

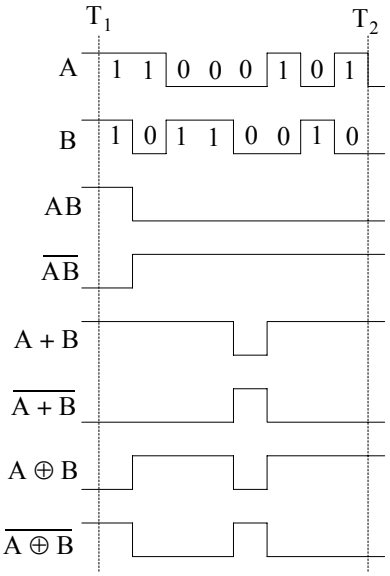


4.

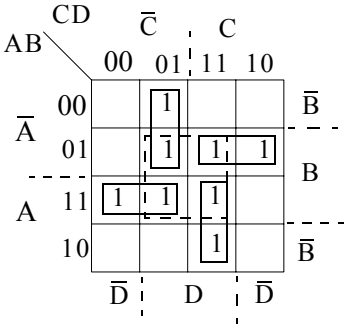


$$W = \overline{\overline{X\bar{Y}}\,\overline{Y\bar{Z}}\,\overline{\bar{X}Z}} = X\bar{Y} + Y\bar{Z} + \bar{X}Z$$

5.



6. The best possible combinations of the variables are shown in the K-map below.



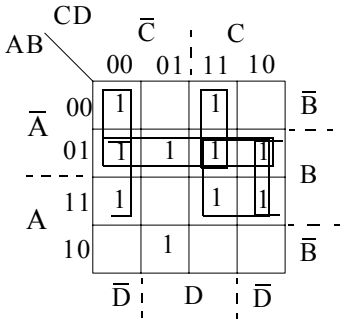
Thus,

$$E = \bar{A}\bar{C}D + \bar{A}BC + AB\bar{C} + ACD$$

The term BD indicated by the dotted rectangle is superfluous since all 1s have already been

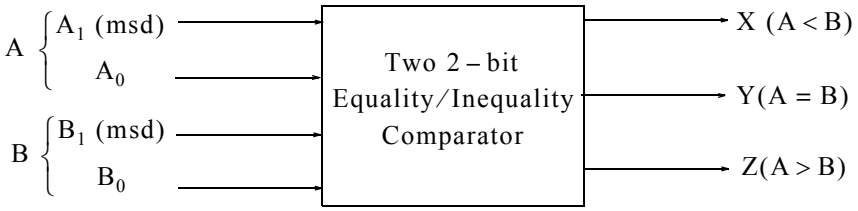
covered by the four terms above.

7. The best possible combinations of the variables are shown below.



$$E = \bar{A}\bar{C}\bar{D} + \bar{A}CD + \bar{A}B + B\bar{D} + BC + A\bar{B}\bar{C}D$$

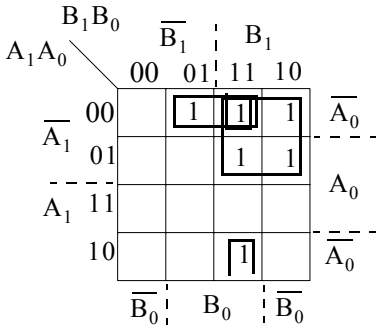
8. The block diagram for this circuit is shown below.



The truth table and the K-maps are shown below.

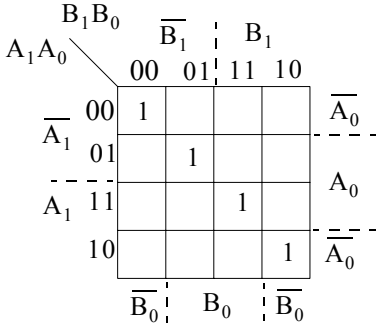
Inputs				Outputs		
A_1	A_0	B_1	B_0	X (A < B)	Y(A = B)	Z(A > B)
0	0	0	0	0	1	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	0	1	0

The K-map for the condition X (A < B) and the corresponding Boolean expression are shown below.



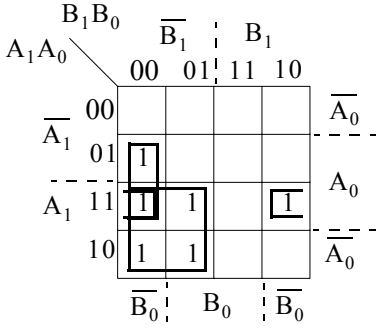
$$X\ (A < B) = \overline{A}_1\overline{A}_0B_0 + \overline{A}_0B_1B_0 + A_1B_1$$

The K-map for the condition Y(A = B) and the corresponding Boolean expression are shown below. See also Example 7.22.



$$X = \overline{A_1 \oplus B_1 + A_0 \oplus B_0}$$

The K-map for the condition Z(A > B) and the corresponding Boolean expression are shown below.



$$Z(A > B) = A_0\overline{B}_1\overline{B}_0 + A_1A_0\overline{B}_0 + A_1\overline{B}_1$$

9. The block diagram of a binary 4-to-2 encoder is shown below.

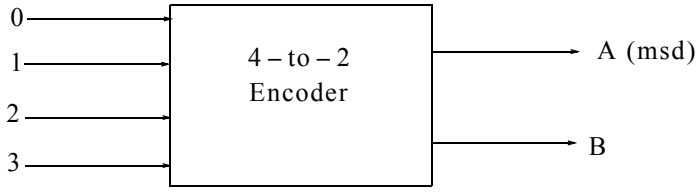


Figure 7.79. Block diagram of 4-to-2 encoder

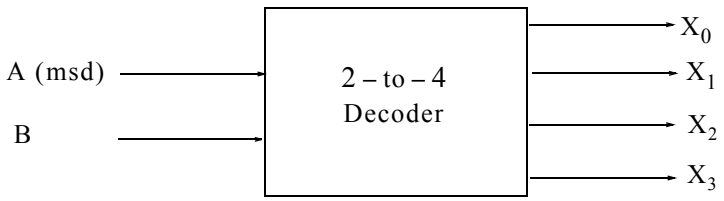
The truth table is shown below where, for convenience, the input lines are shown as D_0 through D_3.

Inputs				Outputs	
D_0	D_1	D_2	D_3	A	B
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

$$A = D_2 + D_3$$

$$B = D_1 + D_3$$

10. The block diagram of a 2-to-4 decoder logic circuit is shown below.



Inputs		Outputs			
A	B	X_0	X_1	X_2	X_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

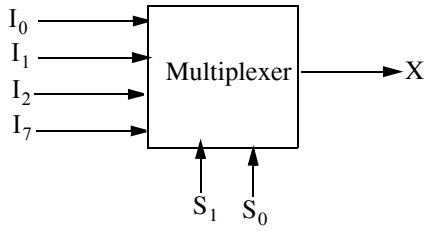
$$X_0 = \overline{A}\,\overline{B}$$

$$X_1 = \overline{A}B$$

$$X_2 = A\overline{B}$$

$$X_3 = AB$$

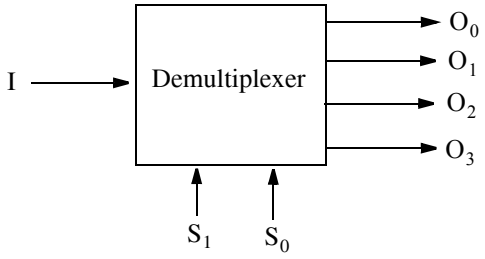
11. The block diagram, truth table, and the Boolean expression for a 4-input digital multiplexer is shown below.



Control lines		Outputs			
S_1	S_0	$X = I_0$	$X = I_1$	$X = I_2$	$X = I_3$
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

$$X = I_0\overline{S_0}\,\overline{S_1} + I_1 S_0\overline{S_1} + I_2\overline{S_0}S_1 + I_3 S_0 S_1$$

12. The block diagram, truth table, and the Boolean expression for a 4-output digital demultiplexer is shown below.



Control lines		Outputs			
S_1	S_0	$I \rightarrow O_0$	$I \rightarrow O_1$	$I \rightarrow O_2$	$I \rightarrow O_3$
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

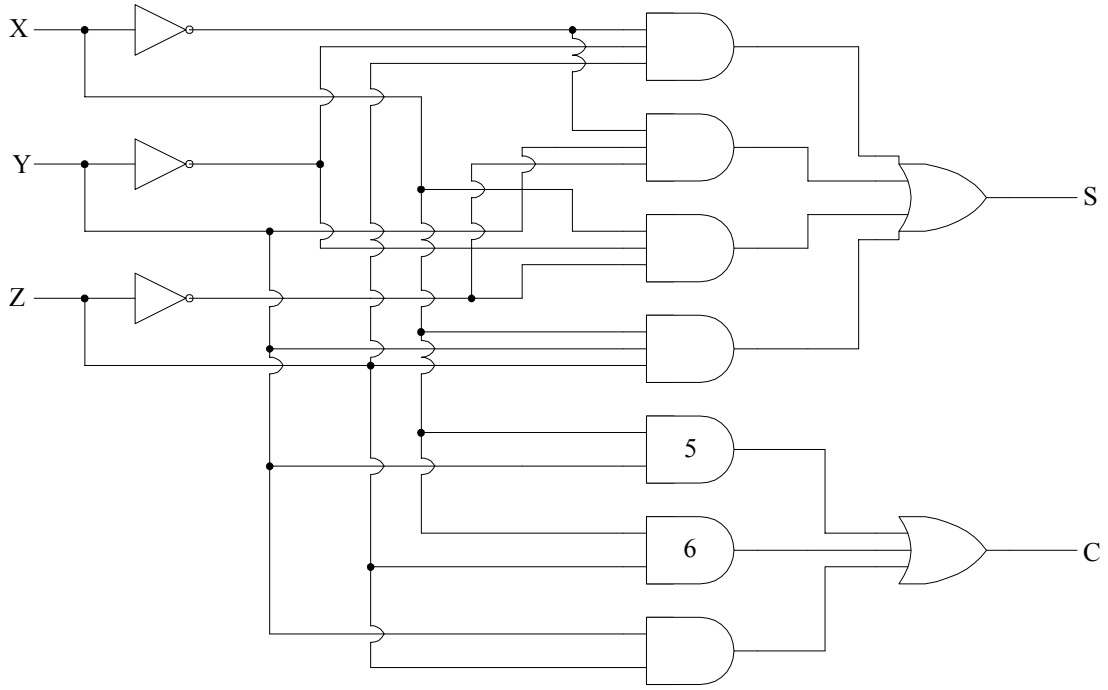
$$O_0 = I\overline{S_0}\,\overline{S_1}$$

$$O_1 = IS_0\overline{S_1}$$

$$O_2 = I\overline{S_0}S_1$$

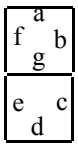
$$O_3 = I\overline{S_0}S_1$$

13. The logic circuit below is that for a full adder and as stated earlier will perform as a full subtractor if we invert the minuend X at the inputs of gates 5 and 6.



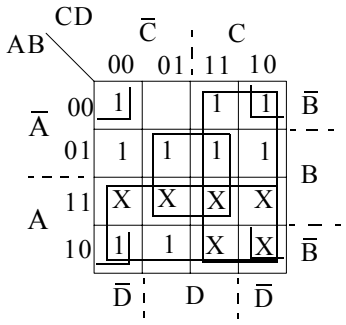
14.

a.



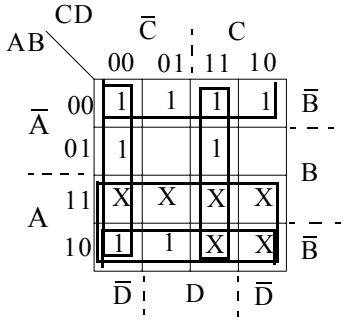
	Inputs (BCD)				Outputs - 7 segments						
Number	A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1
X	1	0	1	0	X	X	X	X	X	X	X
X	1	0	1	1	X	X	X	X	X	X	X
X	1	1	0	0	X	X	X	X	X	X	X
X	1	1	0	1	X	X	X	X	X	X	X
X	1	1	1	0	X	X	X	X	X	X	X
X	1	1	1	1	X	X	X	X	X	X	X

K-map for segment *a* :



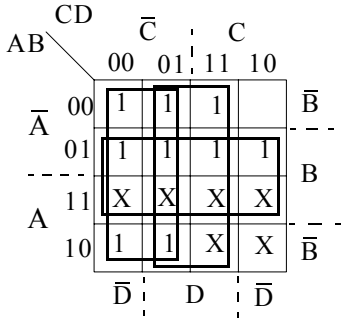
$$a = \overline{B}\overline{D} + BD + A + C$$

K-map for segment *b* :



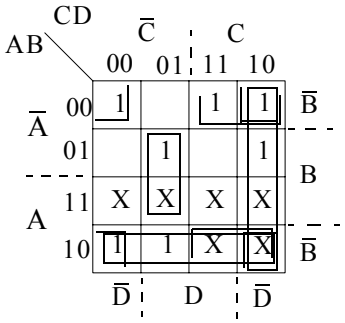
$$b = \overline{B} + \overline{C}\overline{D} + CD + A$$

K-map for segment *c* :



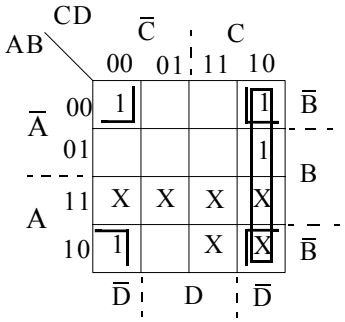
$$c = B + \overline{C} + D$$

K-map for segment *d*:



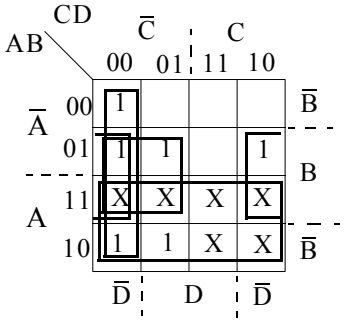
$$d = \bar{B}\bar{D} + \bar{B}C + B\bar{C}D + C\bar{D} + A\bar{B}$$

K-map for segment *e*:



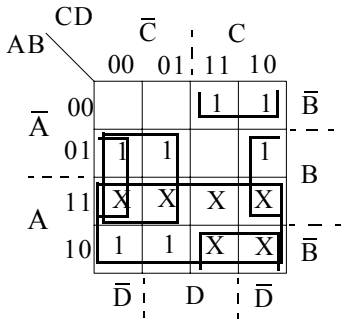
$$e = \bar{B}\bar{D} + C\bar{D}$$

K-map for segment *f*:



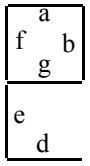
$$f = \bar{C}\bar{D} + B\bar{D} + B\bar{C} + A$$

K-map for segment *g*:

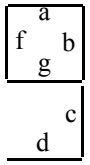


$$g = \bar{B}C + B\bar{D} + B\bar{C} + A$$

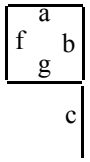
b. For the input 1010, the active (lit) segments are a, b, d, e, f, and g and the display is



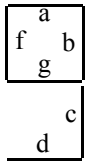
For the input 1011, the active (lit) segments are a, b, c, d, f, and g and the display is



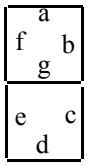
For the input 1100, the active (lit) segments are a, b, c, f, and g and the display is



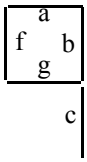
For the input 1101, the active (lit) segments are a, b, c, d, f, and g and the display is



For the input 1110, the active (lit) segments are a, b, c, d, f, and g and the display is



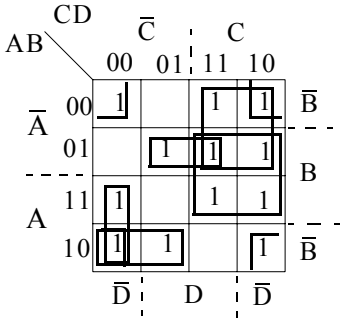
For the input 1111, the active (lit) segments are a, b, c, d, f, and g and the display is



c.

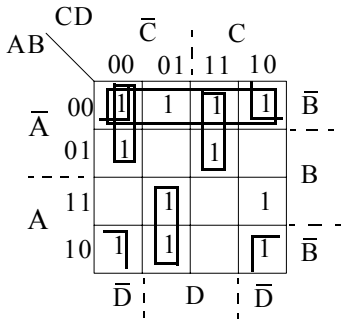
	Inputs (BCD)				Outputs - 7 segments						
Number	A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1
A	1	0	1	0	1	1	1	0	1	1	1
b	1	0	1	1	0	0	1	1	1	1	1
C	1	1	0	0	1	0	0	1	1	1	1
d	1	1	0	1	0	1	1	1	1	0	1
E	1	1	1	0	1	0	0	1	1	1	1
F	1	1	1	1	1	0	0	0	1	1	1

K-map for segment *a* :



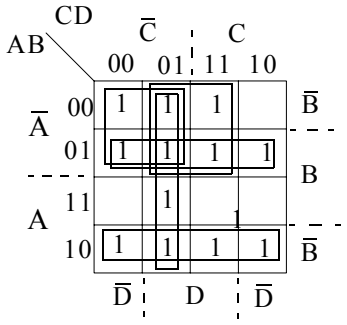
$$a = \bar{B}\bar{D} + A\bar{C}\bar{D} + A\bar{B}\bar{C} + \bar{A}BD + \bar{A}C + BC$$

K-map for segment *b*:



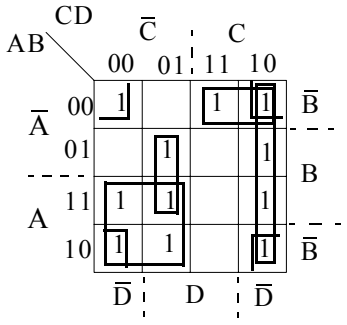
$$b = \bar{B}\bar{D} + \bar{A}\bar{B} + \bar{A}\bar{C}\bar{D} + \bar{A}CD + A\bar{C}D$$

K-map for segment *c*:



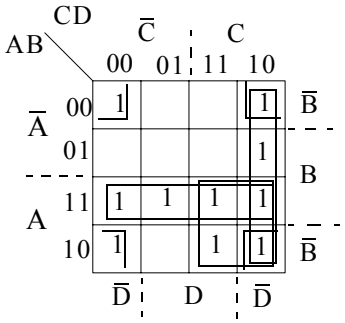
$$c = \bar{A}B + A\bar{B} + \bar{C}D + \bar{A}\bar{C} + \bar{A}D$$

K-map for segment *d*:



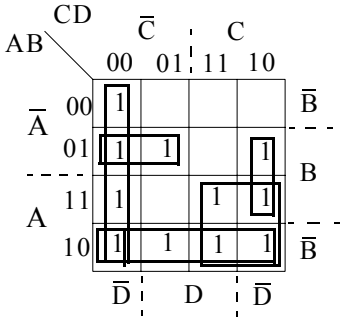
$$d = \bar{B}\bar{D} + A\bar{C} + C\bar{D} + \bar{A}\bar{B}C + B\bar{C}D$$

K-map for segment *e*:



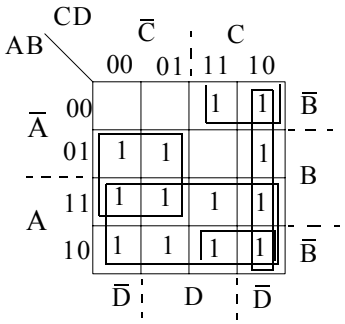
$$e = \bar{B}\bar{D} + AB + AC + C\bar{D}$$

K-map for segment *f*:



$$f = \bar{C}\bar{D} + AC + \bar{A}B\bar{C} + A\bar{B} + BC\bar{D}$$

K-map for segment *g*:



$$g = \bar{B}C + C\bar{D} + B\bar{C} + A$$

Chapter 8

Sequential Logic Circuits

This chapter is an introduction to sequential logic circuits. These are logic circuits where the output(s) depend not only on the input(s) but also on previous states of the output(s). It begins with a discussion of the different types of flip flops, and continues with the analysis and design of binary counters, registers, ring counters, and ring oscillators.

8.1 Introduction to Sequential Circuits

In the previous chapter a combinational logic circuit was defined as a logic circuit whose output(s) are strictly a function of its inputs. A *sequential circuit* is a logic circuit whose output(s) is a function of its input(s) and also its internal state(s). The (internal) state of a sequential logic circuit is either a logic 0 or a logic 1, and because of its ability to maintain a state, it is also called a *memory circuit.* Generally, sequential circuits have two outputs one of which is the complement of the other and multivibrators* fall into this category. Sequential circuits may or may not include logic gates.

Flip flops, also known as *bistable multivibrators*, are electronic circuits with two stable outputs one of which is the complement of the other. The outputs will change only when directed by an input command.

There are 4 types of flip flops and these are listed below.

1. Set-Reset (SR) or Latch
2. D Type (Data or Delay)
3. JK
4. T (Toggle)

8.2 Set-Reset (SR) Flip Flop

Figure 8.21(a) shows a basic *Set-Reset (SR) flip flop* constructed with two NAND gates, and Figure 8.21(b) shows the symbol for the SR flip flop where S stand for Set and R for Reset.

* *For a thorough discussion on multivibrator circuits, please refer to Electronic Devices and Amplifier Circuits, ISBN 0-9744239-4-7*

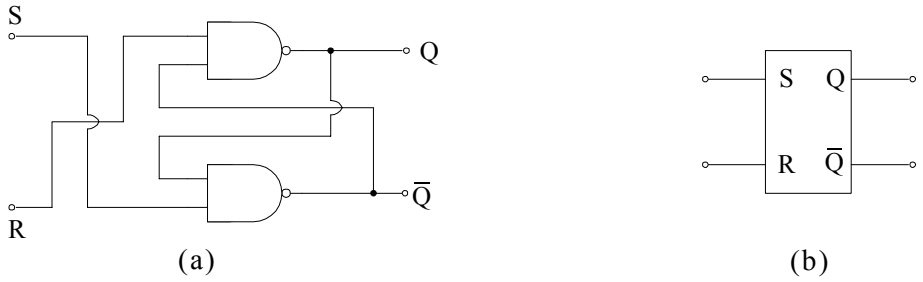


Figure 8.1. Construction and symbol for the SR flip flop

We recall that for a 2-input NAND gate the output is logical 0 when both inputs are 1s and the output is 1 otherwise. We denote the *present state* of the flip flop as Q_n and the *next state* as Q_{n+1}, with reference to Figure 8.1(a) we construct the following *characteristic table*.

TABLE 8.1 Characteristic table for the SR flip flop with NAND gates

Inputs		Present State	Next State		
S	R	Q_n	Q_{n+1}		
0	0	0	1	But $\overline{Q_{n+1}} = 1$ also	The condition where S = R = 0 must be avoided
0	0	1	1	But $\overline{Q_{n+1}} = 1$ also	
0	1	0	0	No Change	
0	1	1	0	Reset (or Clear)	
1	0	0	1	Set	
1	0	1	1	No Change	
1	1	0	0	No Change	
1	1	1	1	No Change	

The characteristic table of Table 8.1 shows that when both inputs S and R are logic 0 simultaneously, both outputs Q and $\overline{Q}$ are logic 1 which is an invalid condition since Q and $\overline{Q}$ are complements of each other. Therefore. the S = R = 0 condition must be avoided during flip flop operation with NAND gates. When R = 1 and S = 0, the next state output Q_{n+1} becomes 0 regardless of the previous state Q_n and this is known as the reset or clear condition, that is, whenever Q = 0 we say that the flip flop is reset or clear. When R= 0 and S = 1, the next state output Q_{n+1} becomes 1 regardless of the previous state Q_n and this is known as the preset or simply set condition, that is. whenever Q = 1 we say that the flip flop is set. When R= 1 and S = 1, the next state output Q_{n+1} remains the same as the present state, that is, there is no state change.

The SR flip flop of Figure 8.1 is an *asynchronous* flip flop, meaning that its inputs and present state alone determine the next state. It is practical, however, to synchronize the inputs of the flip

lop with an external clock and two AND gates in front of the asynchronous flip flop circuit as shown in Figure 8.2. This circuit then becomes a synchronous flip flop and CP is an acronym for Clock Pulse.

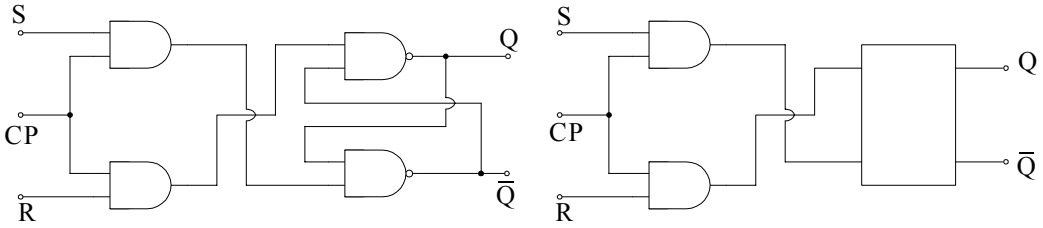


Figure 8.2. Synchronous SR flip flop

An SR flip flop can also be constructed with two NOR gates connected as shown in Figure 8.3.

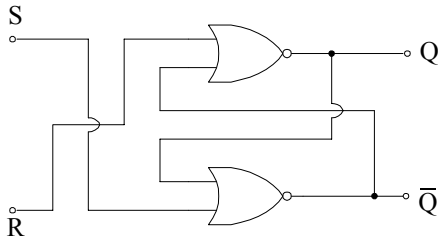


Figure 8.3. SR flip flop with NOR gates

As in the case with the SR flip flop constructed with NAND gates, we use a characteristic table to determine the next states of the SR flip flop with NOR gates shown in Figure 8.23. The characteristic table is shown in Table 8.2.

TABLE 8.2 Characteristic table for the SR flip flop with NOR gates

Inputs		Present State	Next State		
S	R	Q_n	Q_{n+1}		
0	0	0	0	No Change	
0	0	1	1	No Change	
0	1	0	0	No Change	
0	1	1	0	Reset (or Clear)	
1	0	0	1	Set	
1	0	1	1	No Change	
1	1	0	0	But also $\overline{Q_{n+1}} = 0$	The condition where S = R = 1 must be avoided
1	1	1	1	But also $\overline{Q_{n+1}} = 0$	

Table 8.2 shows that when R= 0 and S = 0, the next state output Q_{n+1} remains the same as the present state, that is, there is no state change. When R = 1 and S = 0, the next state output Q_{n+1}

becomes logical 0 regardless of the previous state Q_n and this is known as the reset or clear condition, that is, whenever $Q = 0$ we say that the flip flop is reset or clear. When $R = 0$ and $S = 1$, the next state output Q_{n+1} becomes logical 1 regardless of the previous state Q_n, and this is known as the preset or simply set condition, that is, whenever $Q = 1$ we say that the flip flop is set. When both inputs S and R are logical 1 simultaneously, both outputs Q and $\overline{Q}$ are logical 0 which is an invalid condition since Q and $\overline{Q}$ are complements of each other. Therefore, the $S = R = 1$ condition must be avoided when using NOR gates for the SR flip flop operation.

The SR flip flop with NOR gates shown in Figure 8.3 is an asynchronous flip flop, meaning that its inputs and present state alone determine the next state. It is possible, however, to synchronize the inputs of the flip flop with an external clock and two AND gates in front of the asynchronous flip flop circuit as shown in Figure 8.2 with the NAND gates.

Most flip flops are provided a *set direct* (SD) and a *reset direct* (RD) commands. These commands are also known as SET and CLR (CLEAR). These are asynchronous inputs, that is, they function independently of the clock pulse. Figure 8.4 shows the symbol of an SR flip flop with set direct, reset direct, and clock pulse (CP) indicated by the small triangle.

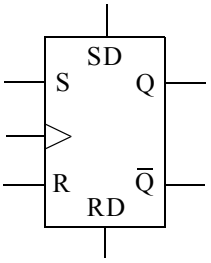


Figure 8.4. Symbol for a typical flip flop with clock pulse, set direct, and reset direct inputs

We should remember that the flip flop inputs are used to tell the flip flop what to do, whereas the clock pulse (CP) tells the flip flop when to do it, and the SD and RD commands bypass the CP command.

8.3 Data (D) Flip Flop

The *data* (D) flip flop is a modification of a synchronous (clocked) SR flip flop the latter of which can be constructed either with NAND or with NOR gates. Figure 8.5 shows the construction of a typical clocked D flip flop with NAND gates, and Table 8.3 its characteristic table.

The characteristic table in Table 8.3 shows that the next state Q_{n+1} is the same as the input D, and for this reason the D flip flop is often called a data transfer flip flop. Its symbol is shown in Figure 8.26 where the small triangle inside the rectangle is used to indicate that this type of flip flop is triggered during the leading edge of the clock pulse and this is normally indicated with the pulse shown in Figure 8.6.

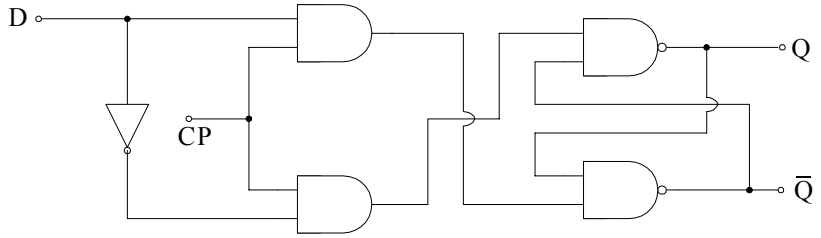


Figure 8.5. Construction of a D flip flop

TABLE 8.3 Characteristic table for the D flip flop with NAND gates

Input	Present State	Next State
D	Q_n	Q_{n+1}
0	0	0
0	1	0
1	0	1
1	1	1

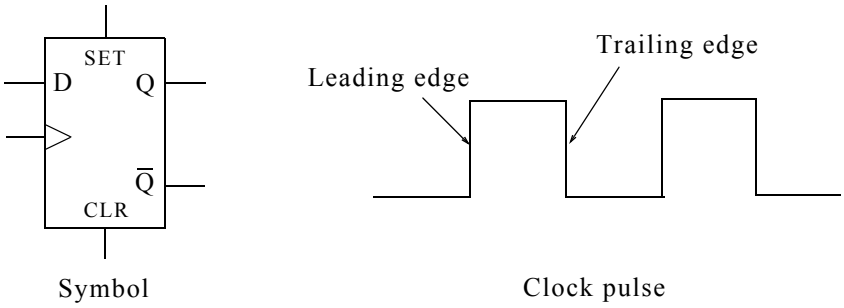


Figure 8.6. D flip flop symbol activated during the leading edge of the clock pulse

Device SN7474 is a D flip flop with asynchronous Set and Clear commands.

8.4 JK Flip Flop

The *JK flip flop* is perhaps the most popular type. Basically, the JK flip flop is a modification of the clocked (synchronous) SR flip flop where one of the two inputs designated as J behaves like an S (set command), and the other input K behaves like an R (reset command). Its most important characteristic is that any combination of the inputs J and K produces a valid output. We recall that in a NAND-gated SR flip flop the input combination S=R=0 must be avoided, whereas in a NOR-gated SR flip flop the input combination S= R= 1 must be avoided. In a JK flip flop however, it is perfectly valid to have J=K=0 simultaneously, or J=K=1 simultaneously.

Figure 8.7 shows is a basic JK flip flop constructed from a basic NOR-gated SR flip flop and its characteristic table is shown in Table 8.4.

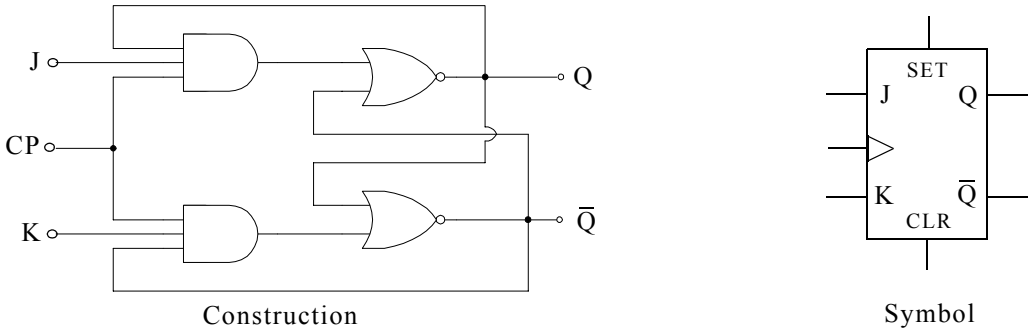


Figure 8.7. Construction of a JK flip flop and its symbol

TABLE 8.4 Characteristic table for the JK flip flop with NOR gates

Inputs		Present State	Next State	
J	K	Q_n	Q_{n+1}	
0	0	0	0	No Change
0	0	1	1	No Change
0	1	0	0	No Change
0	1	1	0	Reset (or Clear)
1	0	0	1	Set
1	0	1	1	No Change
1	1	0	0	Toggle (State change)
1	1	1	1	Toggle (State change)

The characteristic table of Table 8.4 reveals that the JK flip flop behaves as the SR flip flop except that it complements (toggles) the present state output whenever both inputs J and K are logical 1 simultaneously. It should be noted however, that the JK flip flop shown in Figure 8.7 will malfunction if the clock pulse (CP) has a long time duration. This is because the feedback connection will cause the output to change continuously whenever the clock pulse remains High (logic 1) while also at the same time we have J=K=1. To avoid this undesirable operation, the clock pulses must have a time duration that is shorter than the propagation delay through the flip flop. This restriction on the pulse width is eliminated with either the master/slave or the edge triggered JK flip flop which we will discuss shortly.

8.5 Toggle (T) Flip Flop

The *toggle (T) flip flop* is a single input version of the basic JK flip flop, that is, the T flip flop is obtained from the basic JK flip flop by connecting the J and K inputs together where the common point at the connection of the two inputs is designated as T, and it is shown in Figure 8.8 and its characteristic table in Table 8.5.

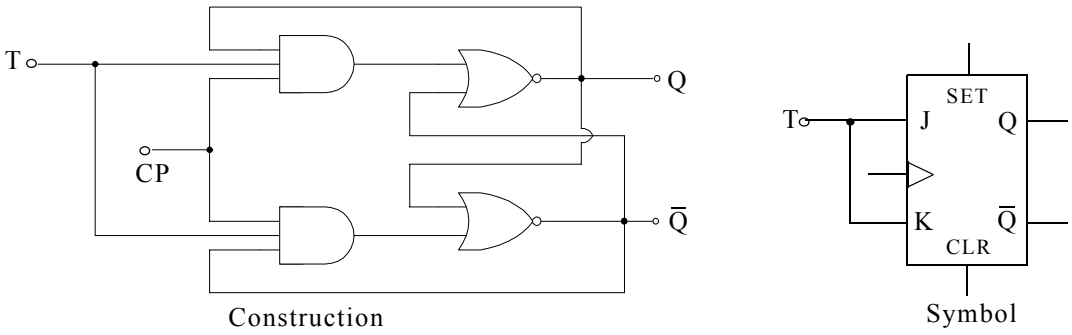


Figure 8.8. Construction of a T flip flop and its symbol

TABLE 8.5 Characteristic table for the T flip flop with NOR gates

Input	Present State	Next State
T	Q_n	Q_{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

The characteristic table of Table 8.5 indicates that the output state changes whenever the input T is logic 1 and the clock pulse is High.

8.6 Flip Flop Triggering

In Section 8.4 we mentioned that a timing problem exists with the basic JK flip flop. This problem can be eliminated by making the flip flop sensitive to pulse transition. This can be explained best by first defining the transition points of a clock pulse. Figure 8.9 shows a *positive pulse* and a *negative pulse*.

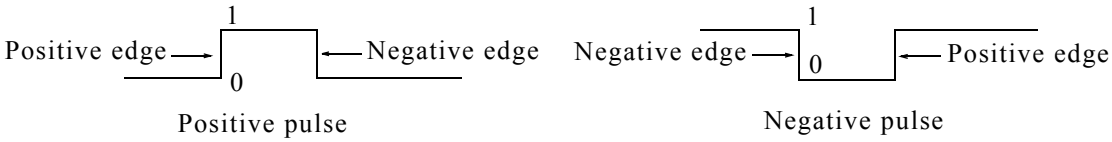


Figure 8.9. Positive and negative pulses

A *positive pulse* is a waveform in which the normal state is logic 0 and changes to logic 1 momentarily to produce a clock pulse. A *negative pulse* is a waveform in which the normal state is logic 1 and changes to logic 0 momentarily to produce a clock pulse. In either case, the *positive edge* is the transition from 0 to 1 and the negative edge is the transition from 1 to 0.

8.7 Edge-Triggered Flip Flops

Edge triggering uses only the positive or negative edge of the clock pulse. Triggering occurs during the appropriate clock transition. Edge triggered flip flops are employed in applications where incoming data may be random. The SN74LS74 IC device shown in Figure 8.10 is a positive edge triggered D type flip flop and the timing diagram of Figure 8.11 shows that the output Q goes from Low to High or from High to Low at the positive edge of the clock pulse.

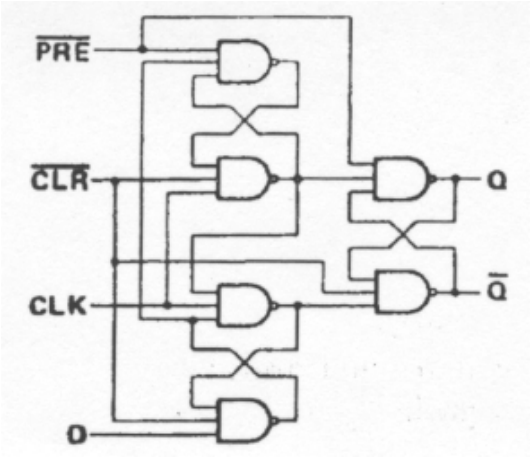


Figure 8.10. The SN74LS74 positive edge triggered D-type flip flop with Preset and Clear (Courtesy Texas Instruments)

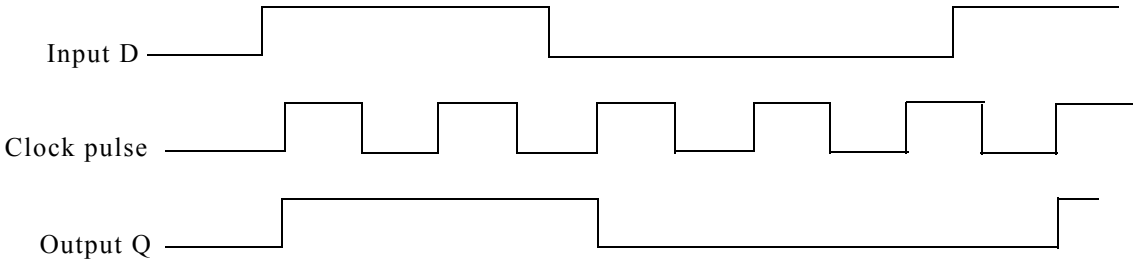


Figure 8.11. Timing diagram for positive edge triggered D flip flop

8.8 Master / Slave Flip Flops

A *master/slave flip flop* consists of two basic clocked flip flops. One flip flop is called the *master* and the other flip flop the *slave*. Figure 8.12 is a block diagram of a master/slave flip flop where the subscript M stands for master and subscript S for slave.

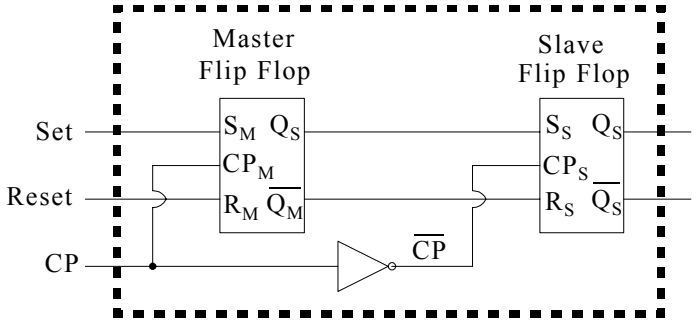


Figure 8.12. Master / Slave flip flop

The master/slave circuit operates as follows:

Whenever the Clock Pulse CP is logic 0 the master flip flop is disabled but the slave flip flop is enabled since $\overline{CP}$ is logic 1. Therefore, $Q_S = Q_M$, and also $\overline{Q_S} = \overline{Q_M}$, that is, the slave flip flop assumes the state of the master flip flop whenever the clock pulse is Low. When the clock pulse is logic High, the master flip flop is enabled and the slave flip flop is disabled. Therefore, as long as the clock pulse is High, the clock pulse input to the slave is disabled and thus the master/slave output Q_S will not change state. Finally, when the clock pulse returns to 0 the master is again disabled, the slave is enabled and assumes the state of the master.

The SN74LS70 IC device is a JK flip flop which is triggered at the positive edge of the clock pulse. Its logic diagram is shown in Figure 8.13 and its symbol is shown in Figure 8.14.

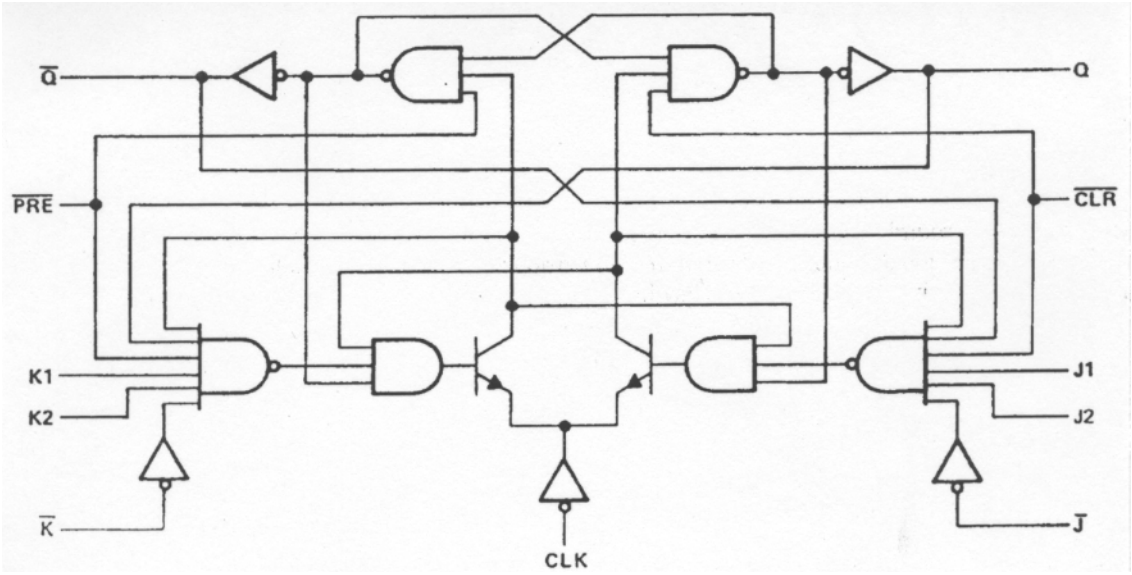


Figure 8.13. Logic diagram for the SN7470 JK flip flop (Courtesy Texas Instruments)

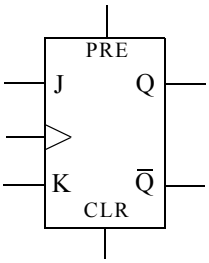


Figure 8.14. Symbol for the SN7470 JK flip flop triggered at the positive edge of the clock pulse

The SN74LS76 IC device is a JK flip flop which is triggered at the negative edge of the clock pulse. Its logic diagram is shown in Figure 8.15 and its symbol is shown in Figure 8.16.

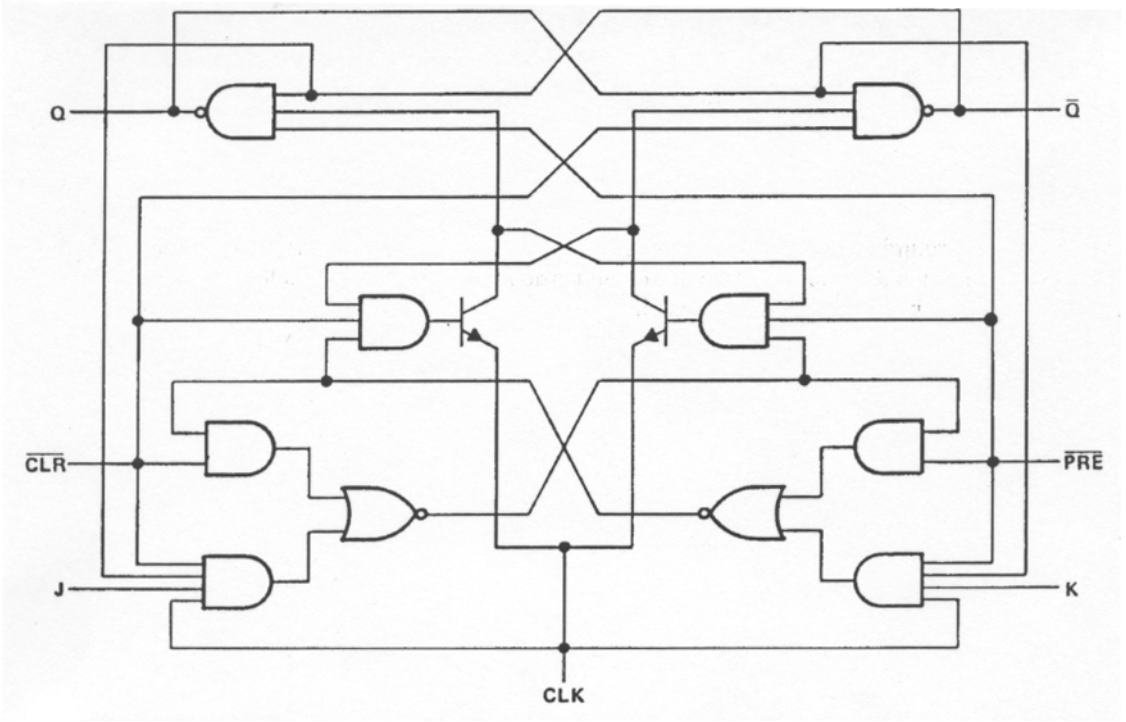


Figure 8.15. Logic diagram for the SN7476 JK flip flop (Courtesy Texas Instruments)

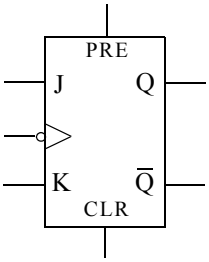


Figure 8.16. Symbol for the SN7476 JK flip flop triggered at the negative edge of the clock pulse

The SN74LS70 JK flip flop is preferable in applications where the incoming data is not synchronized with the clock whereas the SN74LS76 JK flip flop is more suitable for synchronous operations such as synchronous counters which we will be described shortly.

8.9 Conversion from One Type of Flip Flop to Another

By modifying the inputs of a particular type of a flip flop, we can be converted it to another type. The following example illustrates the procedure.

Example 8.1

Construct a clocked JK flip flop which is triggered at the positive edge of the clock pulse from a:

1. clocked SR flip flop consisting of NOR gates

2. clocked T flip flop

3. clocked D flip flop

Solution:

We must determine how the inputs of the given flip flop must be modified so that it will behave as a JK flip flop. We begin with the construction of the *transition* tables shown in Table 8.6 where X denotes a don't care condition.

TABLE 8.6 Transition table for conversion among the four types of flip flops

Flip Flop Type													
SR				JK				D			T		
Q_n	Q_{n+1}	S	R	Q_n	Q_{n+1}	J	K	Q_n	Q_{n+1}	D	Q_n	Q_{n+1}	T
0	0	0	X	0	0	0	X	0	0	0	0	0	0
0	1	1	0	0	1	1	X	0	1	1	0	1	1
1	0	0	1	1	0	X	1	1	0	0	1	0	1
1	1	X	0	1	1	X	0	1	1	1	1	1	0

From the transition table of Table 8.6 we add Columns (1), (2), and (3) to the characteristic table of the JK flip flop as shown in Table 8.7.

Next, we use K-maps to simplify the logic expressions and the modified circuits are shown in Figures 8.17, 8.18, and 8.19.

TABLE 8.7 Characteristic table for the JK flip flop with NOR gates

Inputs		Present State	Next State	(1)		(2)	(3)
J	K	Q_n	Q_{n+1}	S	R	T	D
0	0	0	0	0	X	0	0
0	0	1	1	X	0	0	1
0	1	0	0	0	X	0	0
0	1	1	0	0	1	1	0
1	0	0	1	1	0	1	1
1	0	1	1	X	0	0	1
1	1	0	1	1	0	1	1
1	1	1	0	0	1	1	0

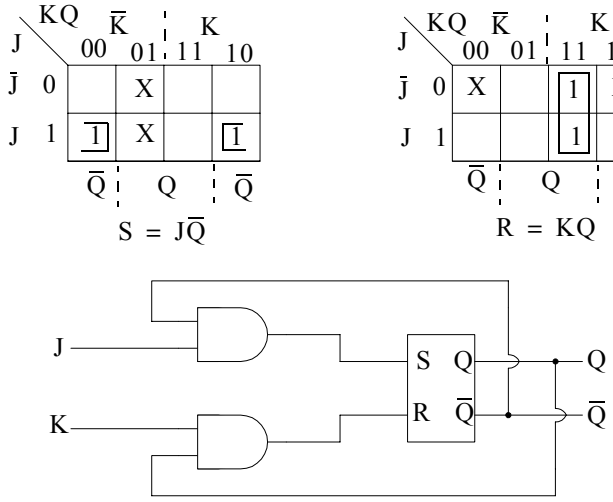


Figure 8.17. SR flip flop converted to JK flip flop

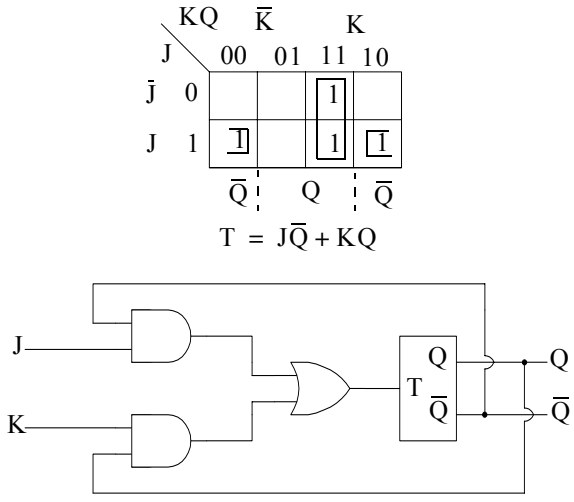


Figure 8.18. T flip flop converted to JK flip flop

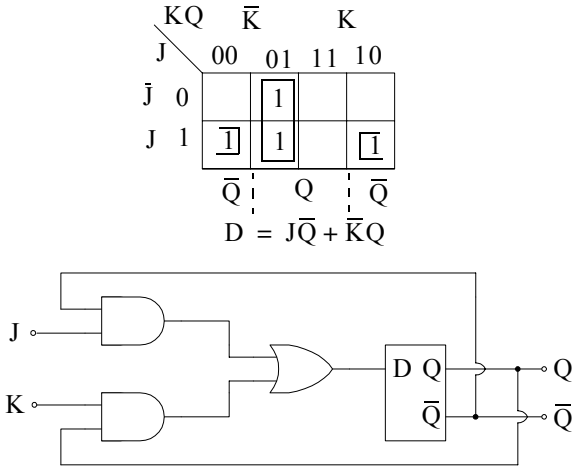


Figure 8.19. D flip flop converted to JK flip flop

8.10 Analysis of Synchronous Sequential Circuits

As mentioned earlier, a sequential circuit contains one or more flip flops and may or may not include logic gates. The circuits in Figure 8.20 are both sequential circuits.

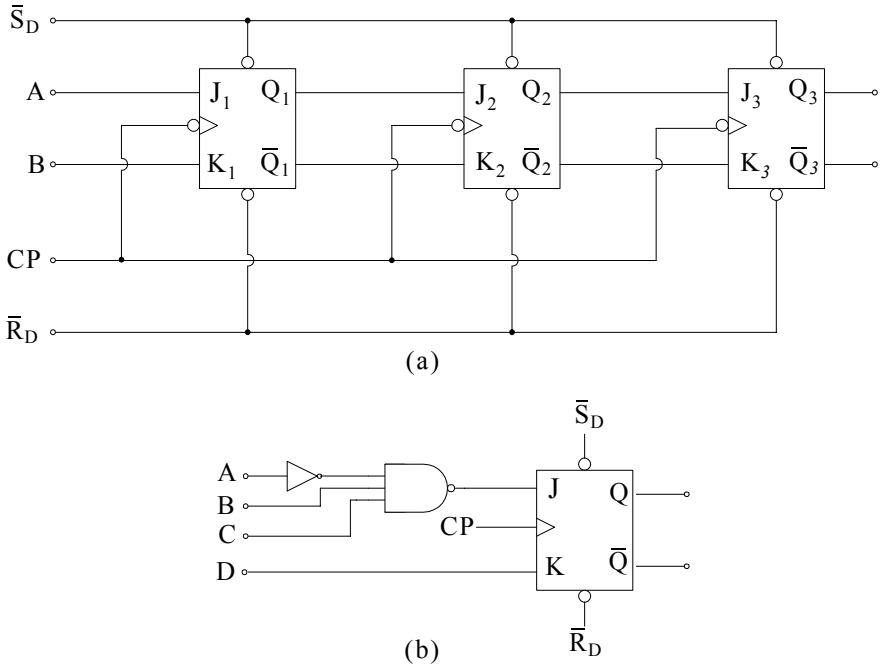


Figure 8.20. Examples of sequential circuits

The circuits of Figures 8.20(a) and 8.20(b) are both *clocked* by the clock pulse *CP*. The circuit of Figure 8.20(a) is also *synchronous* since the clock pulse *CP* is applied simultaneously at all three JK flip flops.

The analysis of synchronous sequential logic circuits is facilitated by the use of *state tables* which consist of three vertical sections labeled present state, flip flop inputs, and next state. The present state represents the state of each flip flop before the occurrence of a clock pulse. The flip flop inputs section lists the logic levels (zeros or ones) at the flip flop inputs which are determined from the given sequential circuit. The next state section lists the states of the flip flop outputs after the clock pulse occurrence. The procedure is illustrated with the following examples.

Example 8.2

Describe the operation of the sequential circuit of Figure 8.21.

Solution:

First, we observe that the JK flip flops are enabled at the negative edge of the clock pulse. Also, the Set Direct $\overline{S}_D$ and Reset Direct $\overline{R}_D$ signals are asynchronous active Low inputs, that is, when they are Low, they override the clock and the data inputs forcing the outputs Q to a logic 1 or logic 0 state. Thus, when $\overline{S}_D$ is active Low, the flip flops will be preset, and when $\overline{R}_D$ is active Low the flip flops will be cleared.

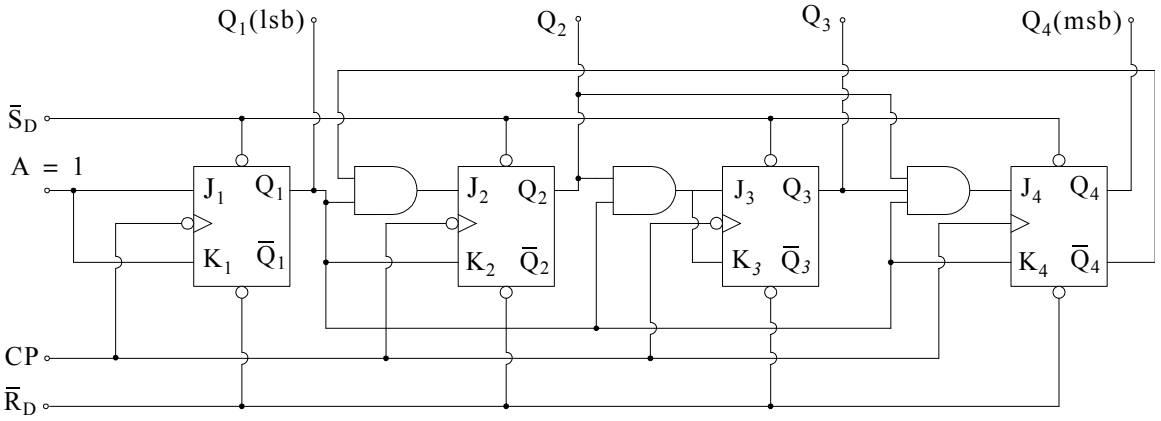


Figure 8.21. Circuit for Example 8.2

Now, we may assume that the given circuit was initially cleared or it was preset, or it may assume any of the sixteen possible states when power is applied to it. In any case, the next state will be determined by the inputs J and K and the previous state of the circuit. Any changes at the outputs Q of each flip flop will take place after the negative edge of the clock pulse has occurred. From the given circuit we see that:

$$\begin{aligned} J_1 &= K_1 = 1 \\ J_2 &= Q_1\overline{Q}_4 \\ K_2 &= Q_1 \\ J_3 &= K_3 = Q_1Q_2 \\ J_4 &= Q_1Q_2Q_3 \\ K_4 &= Q_1 \end{aligned} \qquad (8.1)$$

From the information we have now gathered, we construct the state table shown as Table 8.8.

The Present State section of the above table lists all possible combinations of the outputs Q_1, Q_2, Q_3, and Q_4 before the occurrence of the clock pulse. The Flip Flop Input section is constructed from the equations for J_1, K_1, J_2, K_2, J_3, K_3, J_4, and K_4 shown in relation (8.1). We observe for example, that J_1 and K_1 are always logic 1 so we place 1s under the J_1 and K_1 columns. Likewise, J_2 is logic 1 whenever the combination $Q_1\overline{Q}_4$ is logic 1 and it is zero in all other cases, so we construct the J_2 column as shown in Table 8.8.

TABLE 8.8 State table for Example 8.2

Present State				Flip Flop Inputs								Next State			
Q_4	Q_3	Q_2	Q_1	J_4	K_4	J_3	K_3	J_2	K_2	J_1	K_1	Q_4	Q_3	Q_2	Q_1
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1
0	0	0	1	0	1	0	0	1	1	1	1	0	0	1	0
0	0	1	0	0	0	0	0	0	0	1	1	0	0	1	1
0	0	1	1	0	1	1	1	1	1	1	1	0	1	0	0
0	1	0	0	0	0	0	0	0	0	1	1	0	1	0	1
0	1	0	1	0	1	0	0	1	1	1	1	0	1	1	0
0	1	1	0	0	0	0	0	0	0	1	1	0	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1
1	0	0	1	0	1	0	0	0	1	1	1	0	0	0	0
1	0	1	0	0	0	0	0	0	0	1	1	1	0	1	1
1	0	1	1	0	1	1	1	0	1	1	1	0	1	0	0
1	1	0	0	0	0	0	0	0	0	1	1	1	1	0	1
1	1	0	1	0	1	0	0	0	1	1	1	0	1	0	0
1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1
1	1	1	1	1	1	1	1	0	1	1	1	0	0	0	0

The Next State section is constructed by examination of the Present State and the Flip Flop Input sections with the aid of the JK Flip Flop characteristic table. Thus, the present state with $Q_4 = Q_3 = Q_2 = Q_1 = 0$, becomes the next state with $Q_4 = Q_3 = Q_2 = Q_1 = 1$ after application of the clock pulse because the inputs $J_1 = K_1 = 1$ cause the Flip Flop to toggle, i.e., change state. The information obtained from the state table can be used to construct the state diagram shown in Figure 8.22 where each state of the circuit is represented by a circle and the transition between states is shown by lines interconnecting these circles. The state diagram shows that the states 0000 through 1001 form a ring consisting of these ten states. The given circuit is therefore, a BCD counter that counts from zero to nine, then resets to zero and repeats the counting sequence.

The circuit can be initially cleared with the $\overline{R}_D$ command in which case the circuit will start counting from zero upon application of the clock pulse. Similarly, if the BCD counter is preset with the SD command, the initial state will be 1111 and after application of the first clock pulse the counter will start at state 0000. If neither the preset nor the clear command is used, the counter may assume any state when power is first applied, and that initial state may also be an invalid BCD (1010 through 1111) state. We observe however, that after two clock pulses at most, the counter will enter a valid BCD count and thereafter will enter the zero to nine ring where it will count the normal BCD counting sequence.

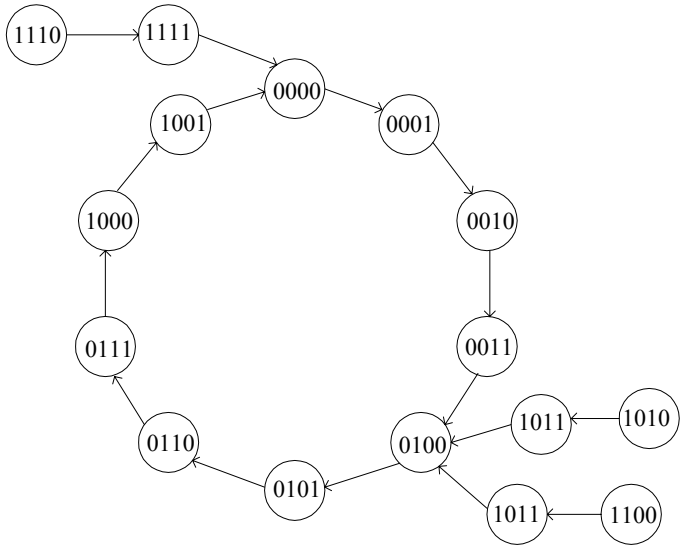


Figure 8.22. State diagram for Example 8.2

Example 8.3

Describe the operation of the sequential circuit of Figure 8.23. As indicated, x is an input that can assume the values of 0 or 1.

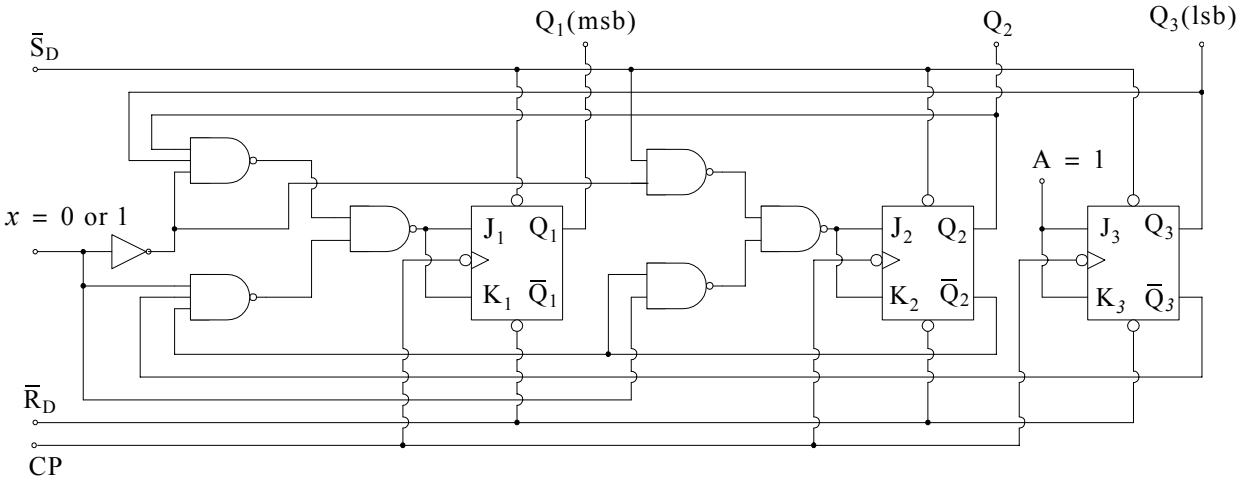


Figure 8.23. Circuit for Example 8.3

Solution:

As with the previous example, we begin with the construction of a state table denoted as Table 8.9. Since the inputs J_1, K_1, J_2, and K_2 are also dependent on the external input x, to facilitate the circuit analysis we divide the Flip Flop Input section of the State Table into two subsections, one for the condition for $x = 0$, and the other for the condition $x = 1$.

TABLE 8.9 State Table for Example 8.3

Present State			Flip Flop Inputs												Next State					
			x=0						x=1						x=0			x=1		
Q_1	Q_2	Q_3	J_1	K_1	J_2	K_2	J_3	K_3	J_1	K_1	J_2	K_2	J_3	K_3	Q_1	Q_2	Q_3	Q_1	Q_2	Q_3
0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1	1	1	1
0	0	1	0	0	1	1	1	1	0	0	0	0	1	1	0	1	0	0	0	0
0	1	0	0	0	0	0	1	1	0	0	1	1	1	1	0	1	1	0	0	1
0	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	0	0	0	1	0
1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	0	1	1
1	0	1	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	1	0	0
1	1	0	0	0	0	0	1	1	0	0	1	1	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	0	1	1	0

The Next State section is also divided into two subsections, one corresponding to the condition $x = 0$ and the other to the condition $x = 1$. From the circuit of Figure 8.23, we obtain the following expressions:

$$\begin{aligned} J_1 &= K_1 = \bar{x}Q_2Q_3 + x\overline{Q_2}\,\overline{Q_3} \\ J_2 &= K_2 = \bar{x}Q_3 + x\overline{Q_3} \\ J_3 &= K_3 = 1 \end{aligned} \tag{8.2}$$

From the state table above, we derive two state diagrams, one for $x = 0$, and the other for $x = 1$ shown in Figure 8.24.

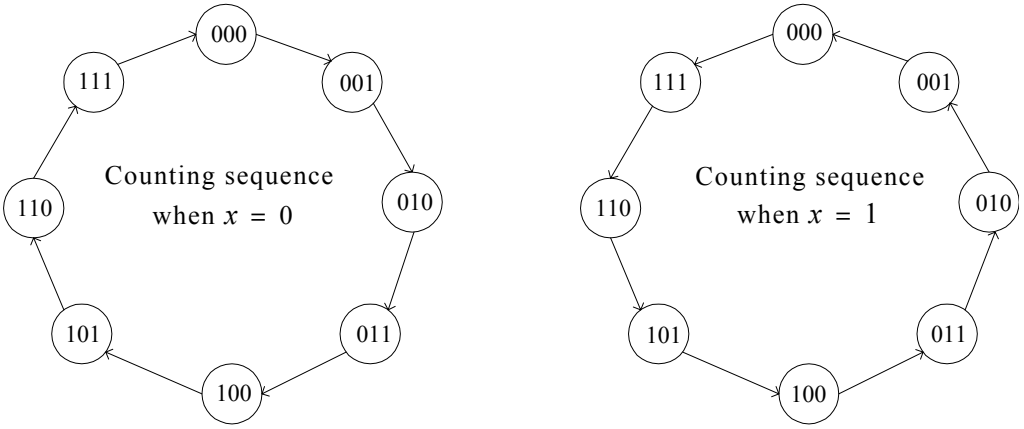


Figure 8.24. State Diagrams for Example 8.3

The state diagrams of Figure 8.24 reveal that the given circuit is an octal counter that counts up from zero to one.... to seven and then repeats whenever $X = 0$, and counts down from seven to six... to zero when $X = 1$. Thus, the circuit is an up/down counter in which the counting

sequence is controlled by the external input X. The given circuit is also provided with Set Direct and Reset Direct commands and thus it has the capabilities of being initially preset or cleared before the counting sequence begins.

Example 8.4

Describe the operation of the sequential circuit of Figure 8.21.

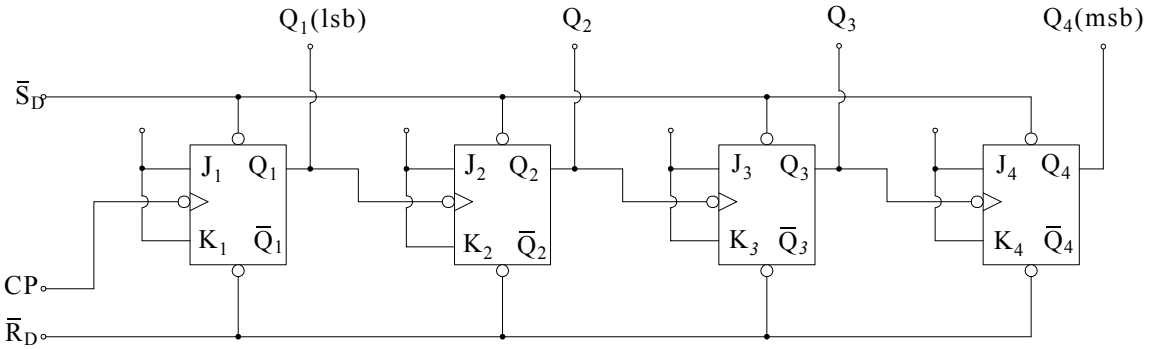


Figure 8.25. Circuit for Example 8.4

Solution:

First, we recall that the small circle at the clock pulse input indicated that the flip flop is triggered (enabled) at the negative edge of the clock pulse. Next, we observe that the clock pulse (CP) signal is connected only to the clock input of the J_1K_1 flip flop. The second flip flop, J_2K_2, is triggered by the Q_1 output of the J_1K_1 flip flop. That is, the Q_1 output serves as the clock pulse input to the J_2K_2 flip flop. Similarly, the Q_2 output serves as the clock pulse input to the J_3K_3 flip flop and the Q_3 output as the clock pulse to the J_4K_4 flip flop.

Now, let us assume that the circuit is initially cleared, i.e., $Q_1 = Q_2 = Q_3 = Q_4 = 0$. Since $J_1 = K_1 = 1$, Q_1 is complemented with each clock pulse count, and since $J_2 = K_2 = 1$ also, every time Q_1 changes from 1 to 0, Q_2 is complemented. Likewise, flip flops J_3K_3 and J_4K_4 change state whenever Q_2 and Q_3 are complemented.

The counting sequence of the given circuit is summarized as shown in Table 8.10. We observe that after the 15th clock pulse has occurred, the outputs of all four flip flops are set so the count is 1111 (decimal 15).On the 16th pulse all four flip flops are reset so the count is 0000 and starts a new counting cycle. This counter is an *asynchronous binary ripple counter*. It is asynchronous be cause not all flip flops are synchronized with the external clock pulse CP. It is called *ripple* because a state change ripples through each flip flop after the external clock pulse CP has been applied. The counting sequence is also obvious from the timing diagram shown in Figure 8.26.

TABLE 8.10 Counting Sequence for the circuit of Figure 8.25

CP	Q_4	Q_3	Q_2	Q_1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1
16	0	0	0	0
17	0	0	0	1

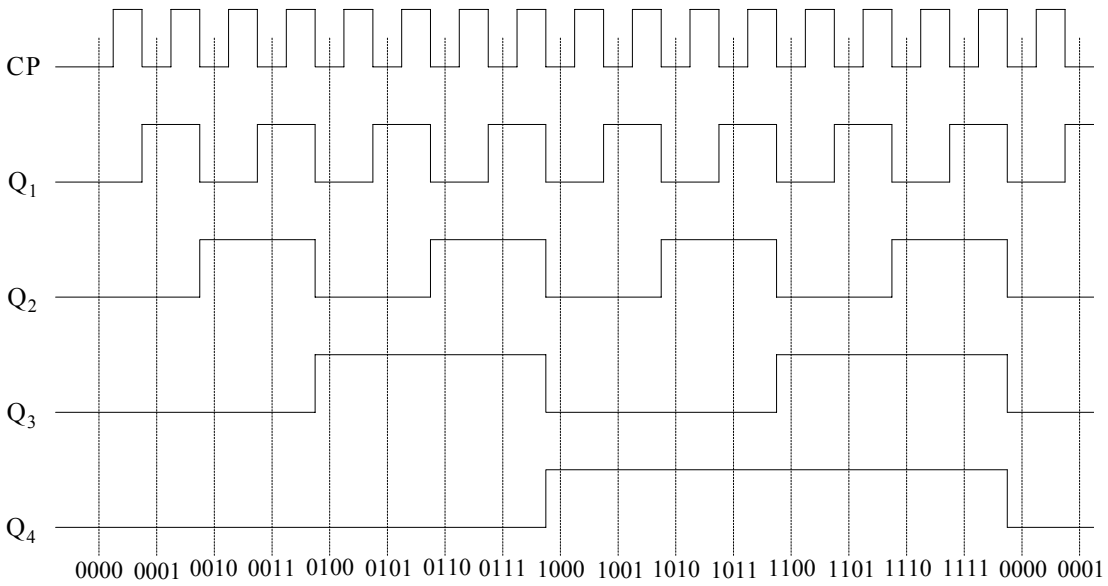


Figure 8.26. Timing diagram for the circuit of Figure 8.25

Two interesting observations are made from the above timing diagram. First, we observe that the e Q_1 output pulse repetition frequency is half of the external clock pulse. Likewise, the Q_2 output pulse repetition frequency is half of the Q_1, or one-fourth of the clock pulse. Similarly, the Q_3 and Q_4 outputs have a pulse repetition frequency which is one eighth and one sixteenth of the external clock pulse respectively. In, other words, the external clock pulse frequency is divided by 2, 4, 8, and 16 when it ripples through the four flip flops. Second, the waveforms of the complements of $\overline{Q}_1$, $\overline{Q}_2$, $\overline{Q}_3$, and $\overline{Q}_4$ will be the inversions of Q_1, Q_2, Q_3, and Q_4 so the complemented waveforms will indicate that the circuit will behave as a down counter when the outputs are taken from the complemented Qs.

Figure 8.27 shows a four-stage (4 flip flop) *ripple-down counter* and Figure 8.28 its timing diagram.

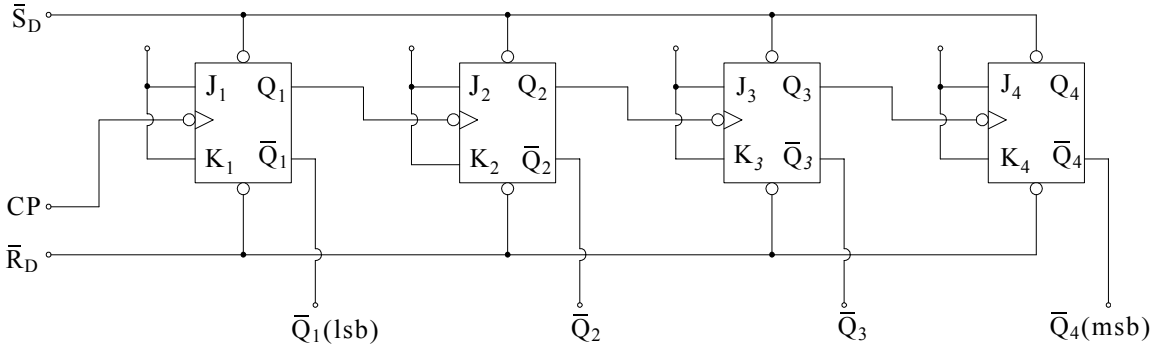


Figure 8.27. Four-stage ripple-down counter

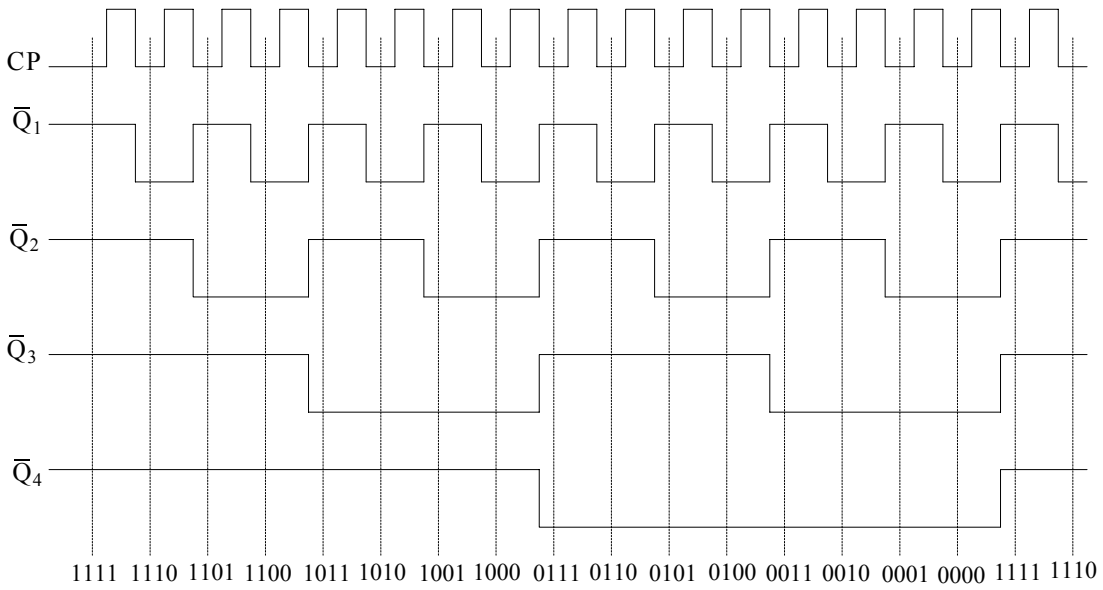


Figure 8.28. Timing diagram for the circuit of Figure 8.27

8.11 Design of Synchronous Counters

The design of synchronous counters is facilitated by first constructing a state diagram and then the state table consisting of the present state, next state, and flip flop inputs sections. The present state and next state sections of the state table are easily constructed from the state diagram. The flip flops input section is derived from the present state and next state sections with the aid of the transition tables that we discussed earlier and are repeated here for convenience.

TABLE 8.11

SR flip flop				JK flip flop				D flip flop			T flip flop		
Q_n	Q_{n+1}	S	R	Q_n	Q_{n+1}	J	K	Q_n	Q_{n+1}	D	Q_n	Q_{n+1}	T
0	0	0	X	0	0	0	X	0	0	0	0	0	0
0	1	1	0	0	1	1	X	0	1	1	0	1	1
1	0	0	1	1	0	X	1	1	0	0	1	0	1
1	1	X	0	1	1	X	0	1	1	1	1	1	0

The design procedure is illustrated with the two examples that follow.

Example 8.5

Design a synchronous BCD up counter, i.e., a counter that will count from 0000 to 0001... to 1001, will return to 0000, and repeat the counting sequence. Use don't care conditions for the illegal BCD combinations (1010 through 1111), and implement the logic circuit with master/slave JK (TTL SN74LS76 device) flip flops and any type of gates.

Solution:

We begin with the construction of the state diagram shown on in Figure 8.29.

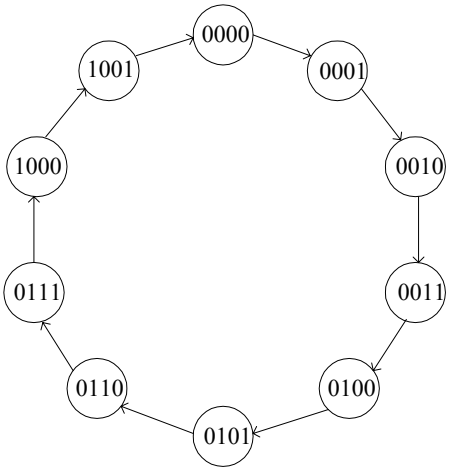


Figure 8.29. State diagram for Example 8.5

The state diagram does not include the states 1010 through 1111 since these will be treated as don't cares. Since our counter will advance through 10 states, we will need four flip flops. We denote Q_4 as the most significant bit (msb) and Q_1 as the least significant bit (lsb).Using the state diagram, we construct the state table shown as Table 8.12.

TABLE 8.12 State table for Example 8.5

Present State				Next State				Flip Flop Inputs							
Q_4	Q_3	Q_2	Q_1	Q_4	Q_3	Q_2	Q_1	J_4	K_4	J_3	K_3	J_2	K_2	J_1	K_1
0	0	0	0	0	0	0	1	0	X	0	X	0	X	1	X
0	0	0	1	0	0	1	0	0	X	0	X	1	X	X	1
0	0	1	0	0	0	1	1	0	X	0	X	X	0	1	X
0	0	1	1	0	1	0	0	0	X	1	X	X	1	X	1
0	1	0	0	0	1	0	1	0	X	X	0	0	X	1	X
0	1	0	1	0	1	1	0	0	X	X	0	1	X	X	1
0	1	1	0	0	1	1	1	0	X	X	0	X	0	1	X
0	1	1	1	1	0	0	0	1	X	X	1	X	1	X	1
1	0	0	0	1	0	0	1	X	0	0	X	0	X	1	X
1	0	0	1	0	0	0	0	X	1	0	X	0	X	X	1
1	0	1	0	X	X	X	X	X	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X	X	X	X	X	X

The Present State of the state table lists all 16 possible states because the circuit may assume any state when power is applied to the circuit. The Next State section shows the outputs of the four flip flops after application of the clock pulse (CP). Thus, if the present state is 0000 the next state will be 0001, and if the present state is 0001 the next state will be 0010, and so on.

The Flip Flop Input section, as mentioned earlier, is constructed with the aid of the JK Flip Flop Transition Table shown in Table 8.11. For example, if the present state is 0100 the next state should be 0101, therefore the Q_1 (lsb) output must be changed from 0 to 1 and according to the transition table for JK flip flop, the JK inputs are assigned the values 1 and X respectively. A similar procedure is used to assign the remaining values at the Flip Flop Input Section. The last six lines of the state table are don't cares and thus we assign X X inputs to all flip flops.

Now, we use the state table to derive expressions for the flip flop inputs as a function of the present state outputs Q_1, Q_2, Q_3, and Q_4. These expressions are simplified by the use of the K-maps shown below. No K-map is needed for J_1 and K_1 since the don't cares of the last two columns of the state table can be treated as logic ones and thus $J_1 = K_1 = 1$.

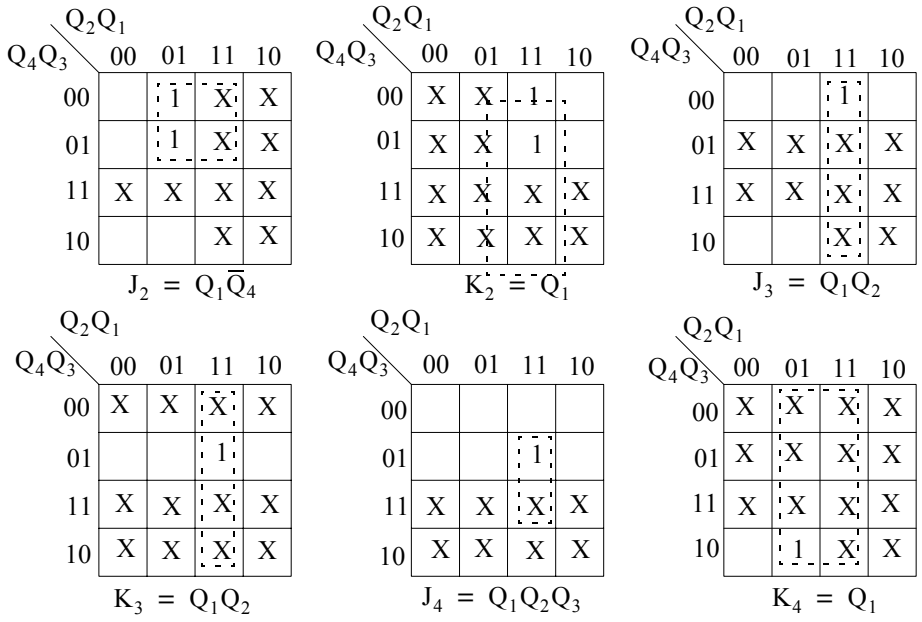


Figure 8.30. K-maps for Table 8.12

We summarize these logic expressions for the flip flop inputs as follows:

$$\begin{aligned} J_1 &= K_1 = 1 \\ J_2 &= Q_1\overline{Q}_4 \\ K_2 &= Q_1 \\ J_3 &= K_3 = Q_1Q_2 \\ J_4 &= Q_1Q_2Q_3 \\ K_4 &= 1 \end{aligned} \tag{8.3}$$

These expressions are the same as those derived from the counter of Example 8.2, relation (8.1). Therefore, our implemented circuit will be identical to that of Example 8.2 which, for convenience, is repeated below as Figure 8.31. This counter can be implemented with two TTL SN74LS76 devices – because each SN74LS76 device contains two Master/Slave Flip Flops with direct preset and direct clear commands – and any types of logic gates. TTL SN74LS160 device is a synchronous 4-bit counter and TTL SN74LS168 is a BCD bi-directional (up/down) counter.

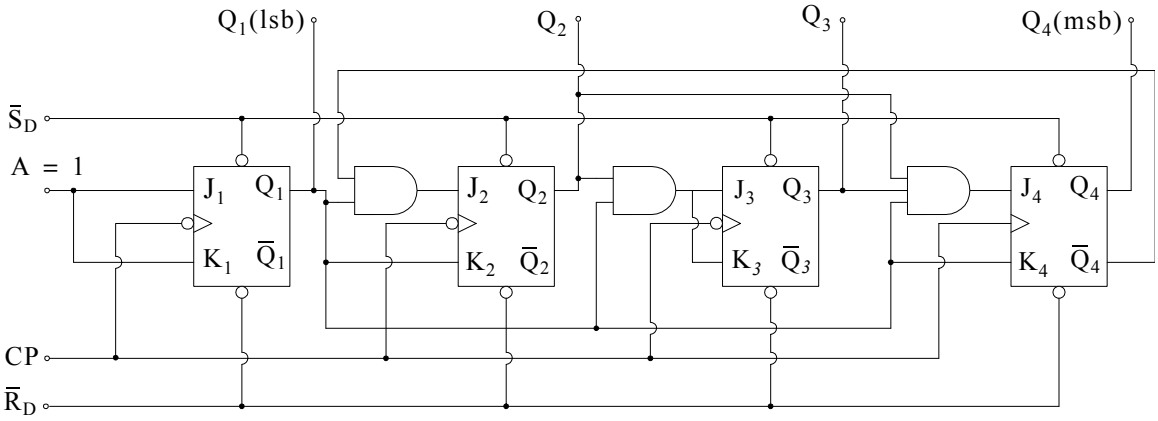


Figure 8.31. Counter for Example 8.5

Example 8.6

Design a counter that counts the decimal digits 0 through 9 represented by the 2*421 code and repeats after the count 9. Use don't care conditions for the unused states and implement the circuit with T-type flip flops and NAND gates only.

Solution:

We discussed the 2*421 code in Chapter 4; it is repeated below for convenience.

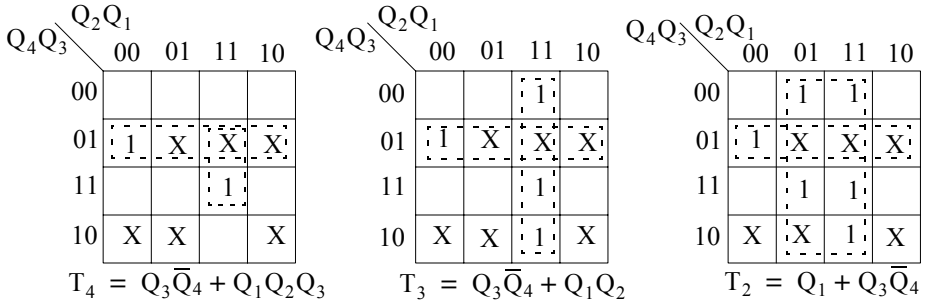
Decimal	2*421
0	0000
1	0001
2	0010
3	0011
4	0100
5	1011
6	1100
7	1101
8	1110
9	1111

As in Example 8.5, we begin a state table shown below as Table 8.13.

TABLE 8.13 State table for Example 8.6

Present State				Next State				Flip Flop Inputs			
Q_4	Q_3	Q_2	Q_1	Q_4	Q_3	Q_2	Q_1	T_4	T_3	T_2	T_1
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	1	0	0	1	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	1
0	0	1	1	0	1	0	0	0	1	1	1
0	1	0	0	1	0	1	1	1	1	1	1
1	0	1	1	1	1	0	0	0	1	1	1
1	1	0	0	1	1	0	1	0	0	0	1
1	1	0	1	1	1	1	0	0	0	1	1
1	1	1	0	1	1	1	1	0	0	0	1
1	1	1	1	0	0	0	0	1	1	1	1
0	1	0	1	X	X	X	X	X	X	X	X
0	1	1	0	X	X	X	X	X	X	X	X
0	1	1	1	X	X	X	X	X	X	X	X
1	0	0	0	X	X	X	X	X	X	X	X
1	0	0	1	X	X	X	X	X	X	X	X
1	0	1	0	X	X	X	X	X	X	X	X

From the state table above we derive the simplest Boolean expressions using the K-maps below.



We summarize these logic expressions for the flip flop inputs as follows:

$$\begin{aligned} T_1 &= 1 \\ T_2 &= Q_1 + Q_3\overline{Q}_4 \\ T_3 &= Q_3\overline{Q}_4 + Q_1Q_2 \\ T_4 &= Q_3\overline{Q}_4 + Q_1Q_2Q_3 \end{aligned} \tag{8.4}$$

With relations (8.4) we implement the circuit shown in Figure 8.32.

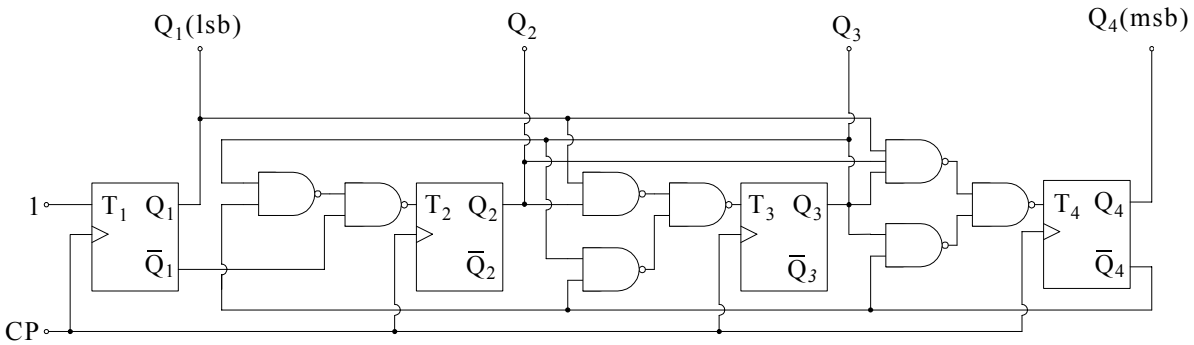


Figure 8.32. Circuit for Example 8.6

8.12 Registers

A *register* is a group of binary storage devices such as flip flops used to store words of binary information. Thus, a group of N flip flops appropriately connected forms an N-bit register capable of storing N bits of information. A register may also contain some combinational circuitry to perform certain tasks.

Transfer of information from one register to another can be either *synchronous transfer* or *asynchronous transfer*. Figure 8.33 shows two examples of synchronous transfer; one with JK flip flops, the other with D flip flops. We observe that in both circuits the second flip flop receives the information from the first via a *transfer command* that can be active Low or active High.

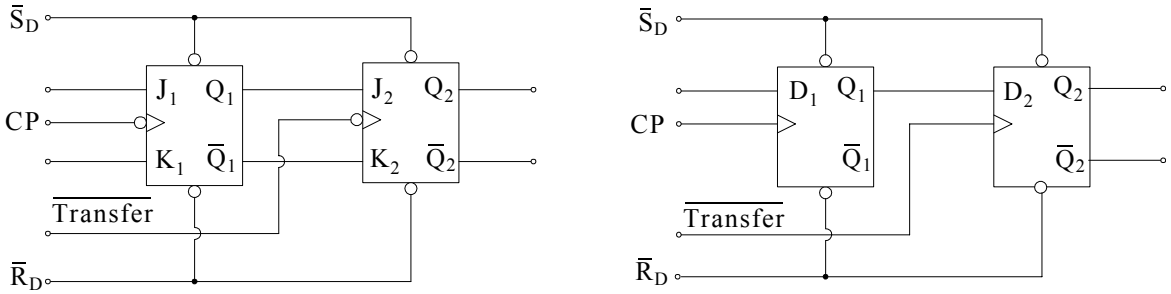


Figure 8.33. Examples of synchronous transfer

An *asynchronous transfer* operation is obtained when the transfer (active Low or active High) command is connected to the $\overline{S}_D$ and/or the $\overline{R}_D$ inputs of a clocked flip flops as in Figure 8.34.

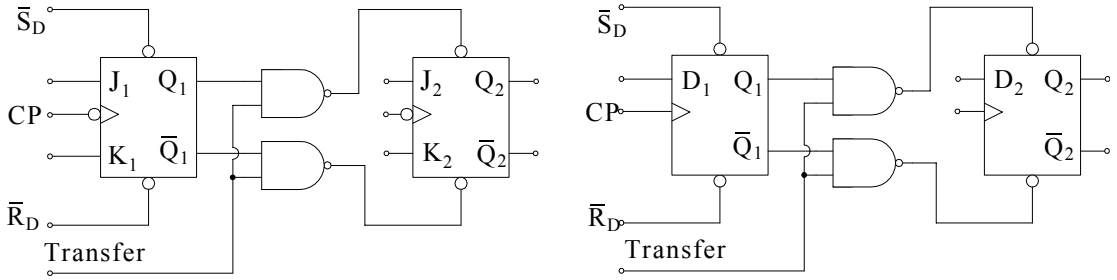


Figure 8.34. Examples of asynchronous transfer

Binary words can be transferred from one register to another either in parallel or serial mode. In a *parallel transfer* all bits of the binary word are transferred simultaneously with a single transfer command.

Example 8.7

Registers A and B contain five D-type flip flops each triggered at the positive edge of the clock pulse. Design a digital circuit that will transfer the data from Register A to Register B with a single transfer command.

Solution:

Figure 8.35 shows how the data from Register A is transferred to Register B with a single transfer command.

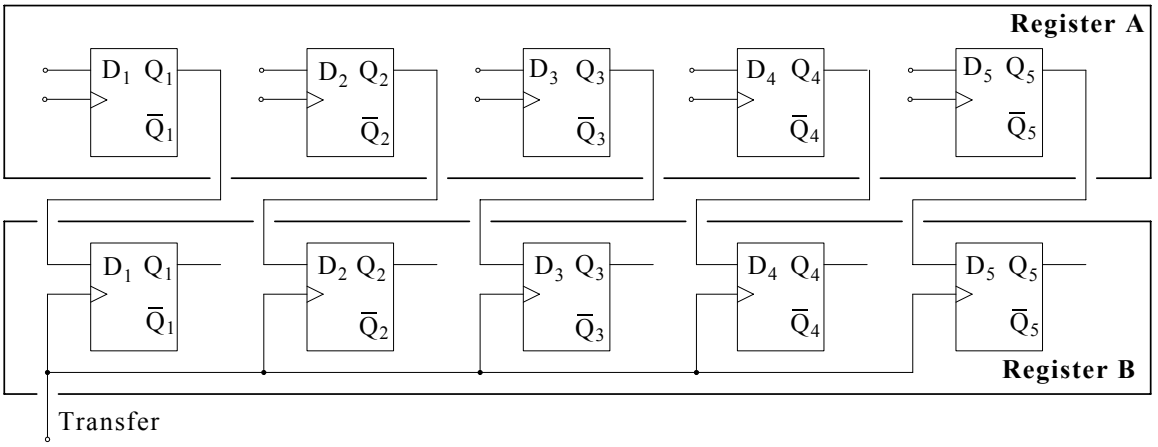


Figure 8.35. Parallel transfer of binary words from one register to another for Example 8.7

Some registers, like the one shown in Figure 8.36, are provided with an additional command line referred to as the *read command* in addition to the transfer command which is often referred to as the *load command*.

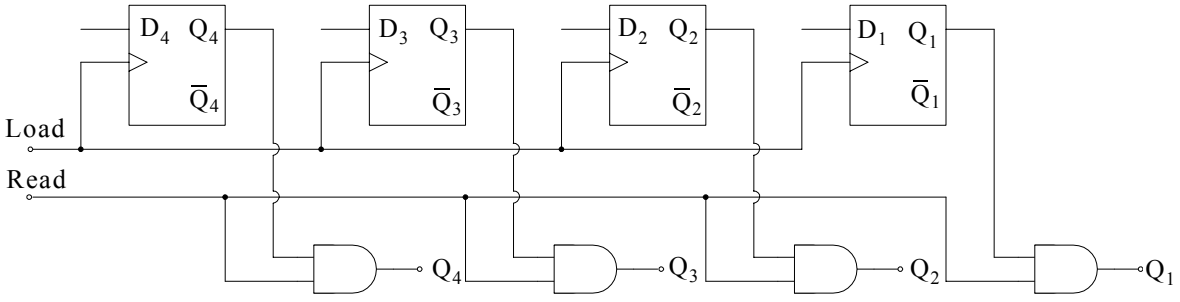


Figure 8.36. Register with load and read commands

In Figure 8.36, the load command allows the register to receive data that may come from any source and are fed into the D_1 through D_4 inputs of these flip flops. After the data are loaded or stored, the read command may be used to transfer that data to another register, and even though the data may have been transferred to another register, they have not been lost at the source register.

A *shift register* is a register that is capable of shifting the bits of information stored in that register either to the left or to the right. Either the input or the output of a shift register may be serial or parallel. A register with a serial input is one in which the incoming data are received one bit at a time, where the least significant bit of the incoming binary word is received first and the most significant bit (msb) is received last. After the lsb of the incoming data is stored in the first flip flop of the register, it is shifted to the next flip flop to the right of the first one so that the first flip flop can receive the next arriving bit.

A shift register can be a left-shift register, a right-shift register, a serial-to-parallel register, a parallel-to-serial register, or a parallel input/parallel output register. The register shown in Figure 8.37 is an example of serial transfer of data between Registers A and B. We observe that both registers are right-shift registers and therefore the contents of Register A are transferred to Register B after three shift pulses.

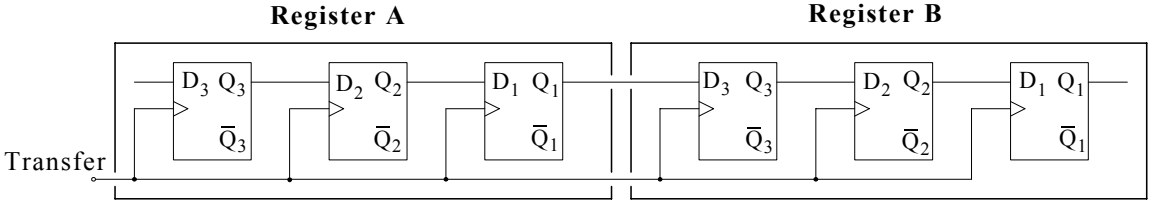


Figure 8.37. An example of serial transfer of data between two registers

Figure 8.38 shows a serial-to-parallel register where the reset line $\overline{R}_D$ is used to clear the register before the incoming data. Then, the serial input is stored one bit at a time into the register. After the data is stored in the register, it can be read from the parallel output lines Q_1 through Q_4.

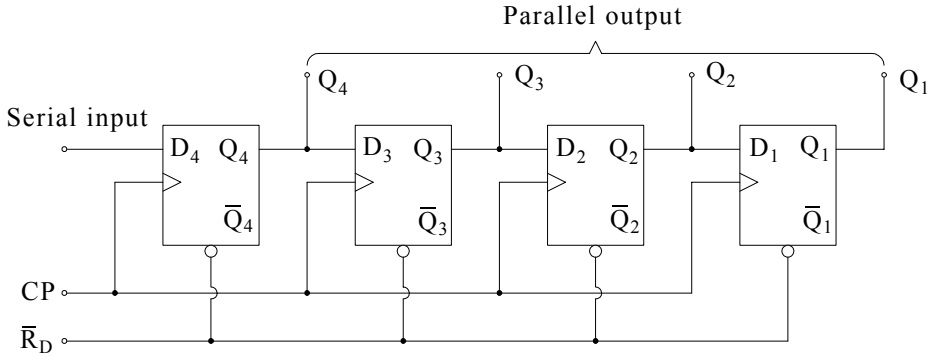


Figure 8.38. An example of serial input, parallel output register

The register of Figure 8.39 is an example of a parallel-to-serial register. We observe that the input

data are available at the Q_1 through Q_4 input lines which represent the outputs of another register. The reset line $\overline{R}_D$ is used to clear the flip flops of the register. The load command stores the data in a parallel mode asynchronously. When the clock pulse (CP) signal is applied, the stored data is shifted to the right, and since $J_4 = 0$ and $K_4 = 1$, the register will be cleared after four clock pulses and it will be ready to receive the next incoming binary data.

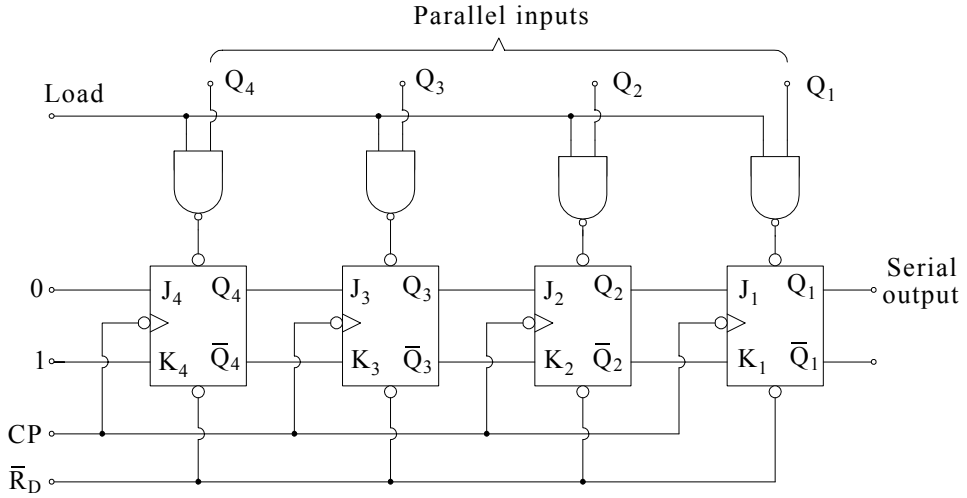


Figure 8.39. An example of parallel input, serial output register

Example 8.8

Design a parallel input/parallel output register consisting of four D-type flip flops triggered at the positive edge of the clock pulse. Make provisions that the register can shift the data either to the left or to the right.

Solution:

The register shown in Figure 8.40 meets the design specifications. The register should be first cleared by the $\overline{R}_D$ command. Next, the load command transfers the incoming data into the register. The data are read at the output lines whenever the read command is activated. The. register contents can be shifted to the left or to the right by the left or right shift commands. The delay element is used to allow the data to reach the D_1 through D_4 inputs of the flip flops before the clock pulse signal is enabled. The exclusive-OR gate ensures that the left and right shift commands are mutually exclusive, that is, they should not be activated simultaneously.

Figure 8.41 shows the commercially available SN74LS194 4-bit bidirectional universal shift register. The mode control signals S0 and S1are used to set four distinct modes of operation as shown in Table 8.14. The maximum clock frequency is 36 MHz and the power consumption is 75 mW.

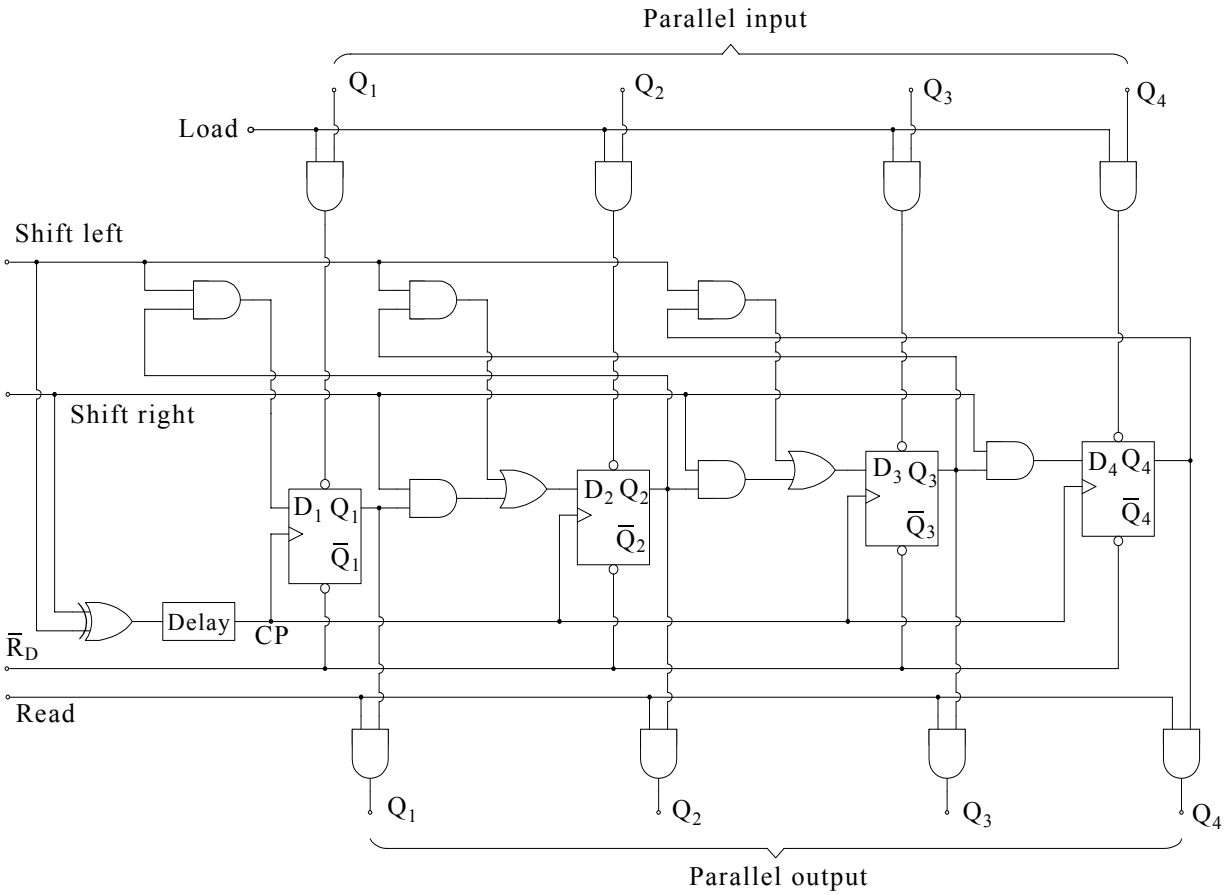


Figure 8.40. Parallel input, parallel output register for Example 8.8

TABLE 8.14 Modes of operation for the SNLS194 4-bit bidirectional shift register

S1	S0	Operation
0	0	Do nothing - inhibit clock
0	1	Right shift - in the direction Q_A toward Q_D
1	0	Left shift - in the direction Q_D toward Q_A
1	1	Synchronous parallel load

As shown in Table 8.14, clocking of the shift register is inhibited when both mode control inputs S0 and S are Low, and the other three modes of operation are changed only while the clock input is High. Shift right is accomplished synchronously with the positive (rising) edge of the clock pulse when S0 is High and S1 is Low. When S0 is Low and S1 is High, the data are shifted left synchronously and new data is entered at the shift left serial input. Synchronous parallel loading is achieved when both S0 and S1 are High. The data are loaded into the appropriated flip flops and appear at the outputs after the positive transition of the clock input. During loading, serial data flow is inhibited.

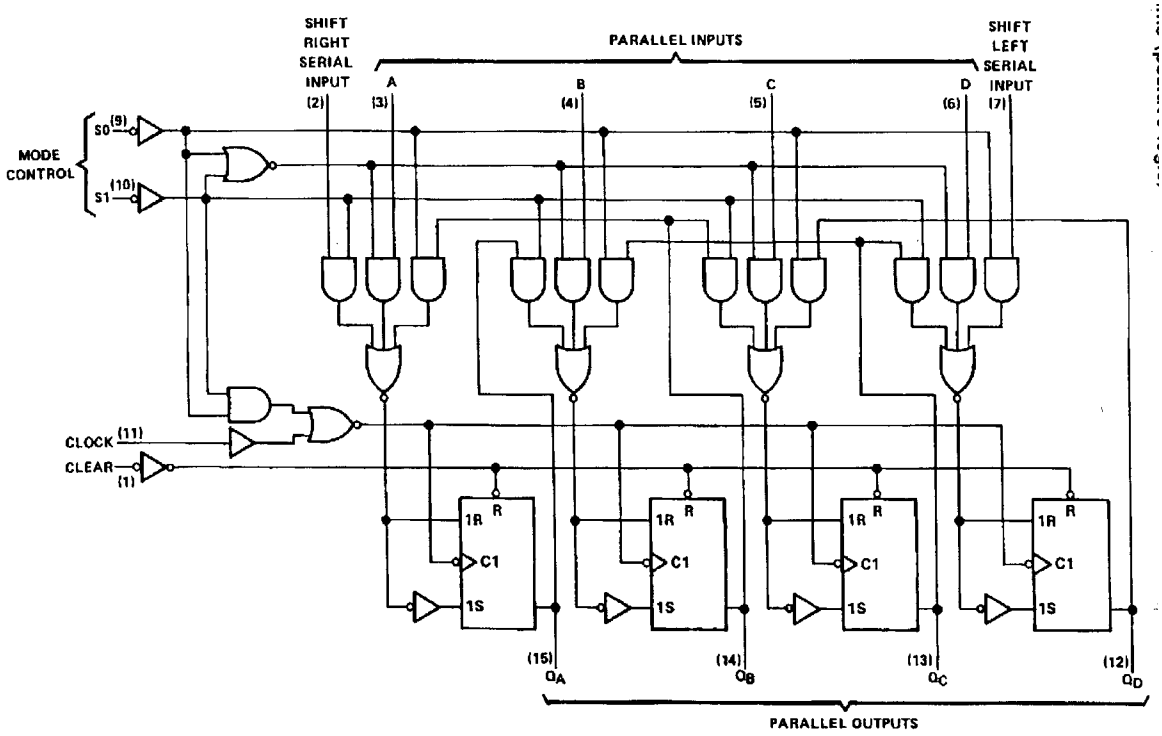


Figure 8.41. The SN74LS194 4-bit bidirectional universal shift register (Courtesy Texas Instruments)

8.13 Ring Counters

Ring counters are a special type of counters that can be used to generate timing signals. A 4-bit ring counter is shown in Figure 8.42, the generated waveforms are shown in Figure 8.43, and its state table is shown in Table 8.15.

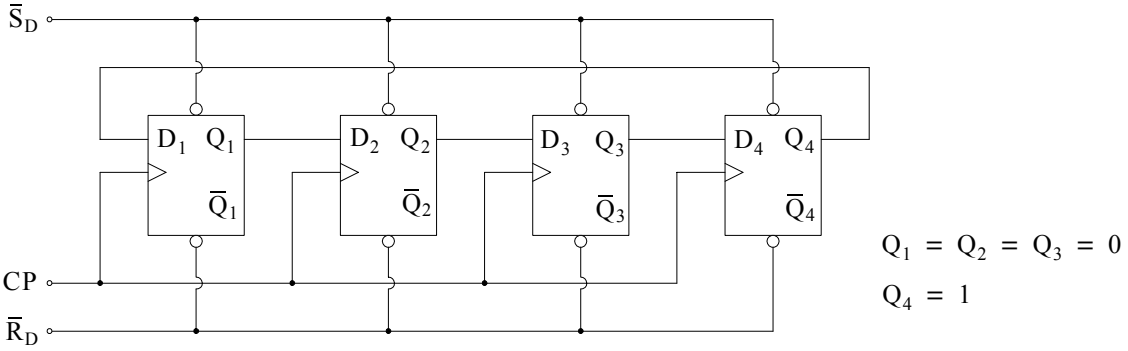


Figure 8.42. A 4-bit ring counter

For the ring counter of Figure 8.42 it is assumed that flip flops D_1 through D_3 are initially reset to 0 and flip flop D_4 is preset to logic 1. This is a mandatory condition for any ring counter, that is, at least one flip flop of the ring counter must be preset to 1 and at least one flip flop must be reset

to 0, otherwise the ring counter has no meaning since none of its outputs will change state in this case.

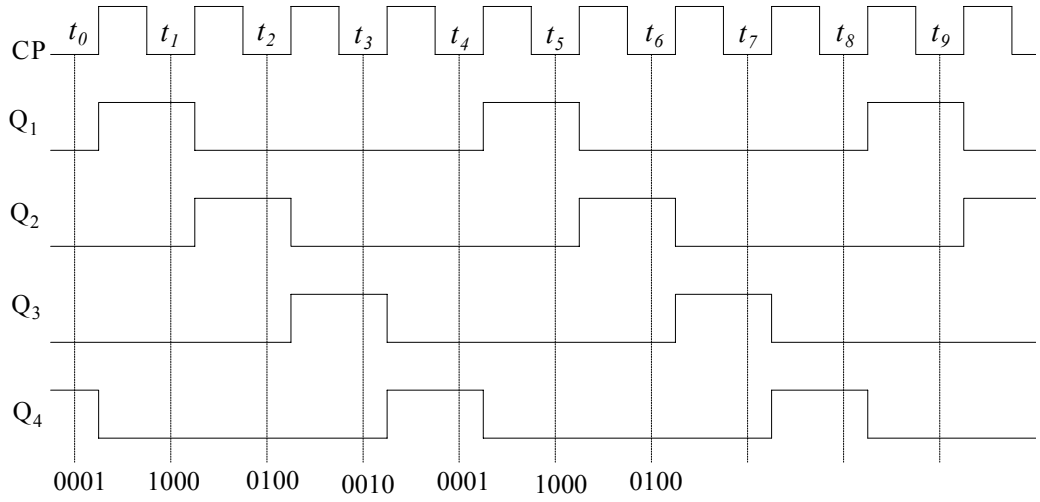


Figure 8.43. Timing diagram for the ring counter of Figure 8.42

TABLE 8.15 Counting sequence for the ring counter of Figure 8.42

Time Count		State			
		Q_1	Q_2	Q_3	Q_4
t_0	0	0	0	0	1
t_1	1	1	0	0	0
t_2	2	0	1	0	0
t_3	3	0	0	1	0
t_4	4	0	0	0	1
t_5	5	1	0	0	0
and so on					

The *Johnson counter* is a modified ring counter where the complemented output of the last flip flop is connected to the input of the first flip flop. Figure 8.44 shows a 4-stage Johnson counter, Figure 8.45 shows its timing diagram, and Table 8.16 shows its state table.

There are two basic differences between a conventional ring counter and a Johnson counter. First, a Johnson counter may start with all its flip flops in the reset condition, whereas a conventional ring counter requires at least one flip flop preset to logic 1 and at least one flip flop reset to logic 0. Second, a Johnson counter generates $2n$ timing signals with n flip flops whereas a conventional ring counter generates n timing signals with n flip flops. For example, to generate 8 timing signals, one would require a ring counter with 8 flip flops whereas a Johnson counter with 4 flip flops will suffice as we can see from Tables 8.15 and 8.16.

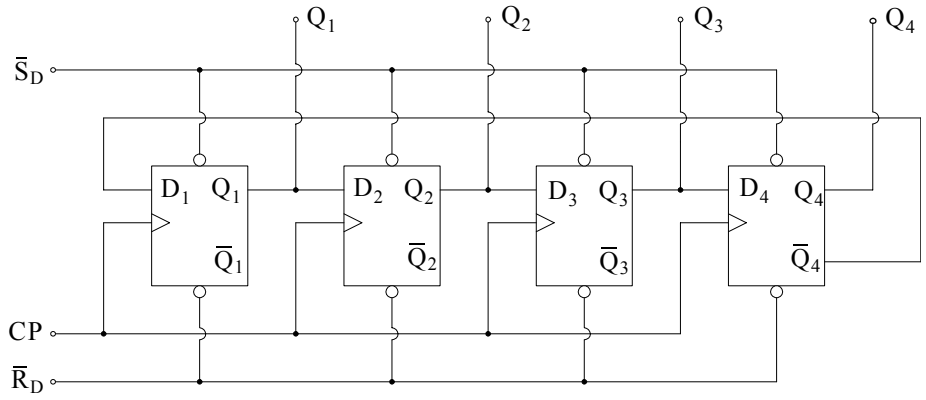


Figure 8.44. A 4-bit Johnson counter

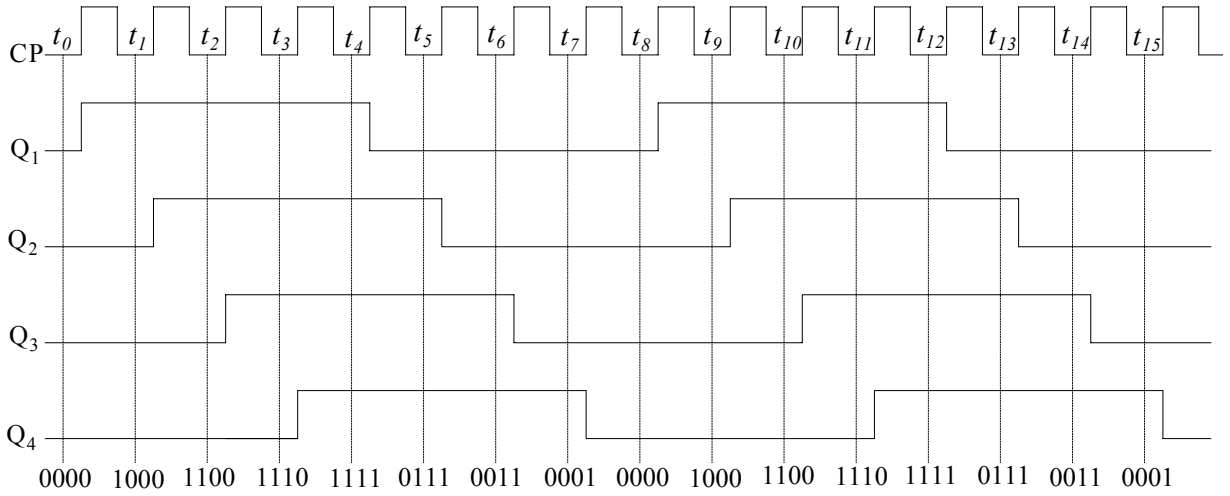


Figure 8.45. Timing diagram for the Johnson counter of Figure 8.44

TABLE 8.16 Counting sequence for the ring counter of Figure 8.42

Time Count		State			
		Q_1	Q_2	Q_3	Q_4
t_0	0	0	0	0	0
t_1	1	1	0	0	0
t_2	2	0	1	0	0
t_3	3	0	0	1	0
t_4	4	0	0	0	1
t_5	5	1	0	0	0
and so on as we can see in Figure 8.45					

8.14 Ring Oscillators

The ring oscillator* is an interesting arrangement of a ring consisting of an odd number of inverters, such as 3, 5, 7, and so on in a loop, and it is used in digital circuits. Figure 8.46 shows a ring oscillator in a ring of 5 inverters, and Figure 8.47 shows the resulting timing diagrams where it is assumed that before the first CP occurs, the output of the 5th inverter is logic 1. It is also assumed that the inverters are identical and are triggered at the positive edge of the clock pulse. The period of each clock pulse represents the time delay of each inverter.

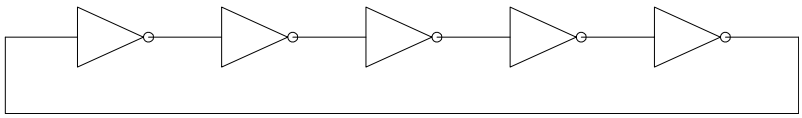


Figure 8.46. A 5-inverter ring oscillator

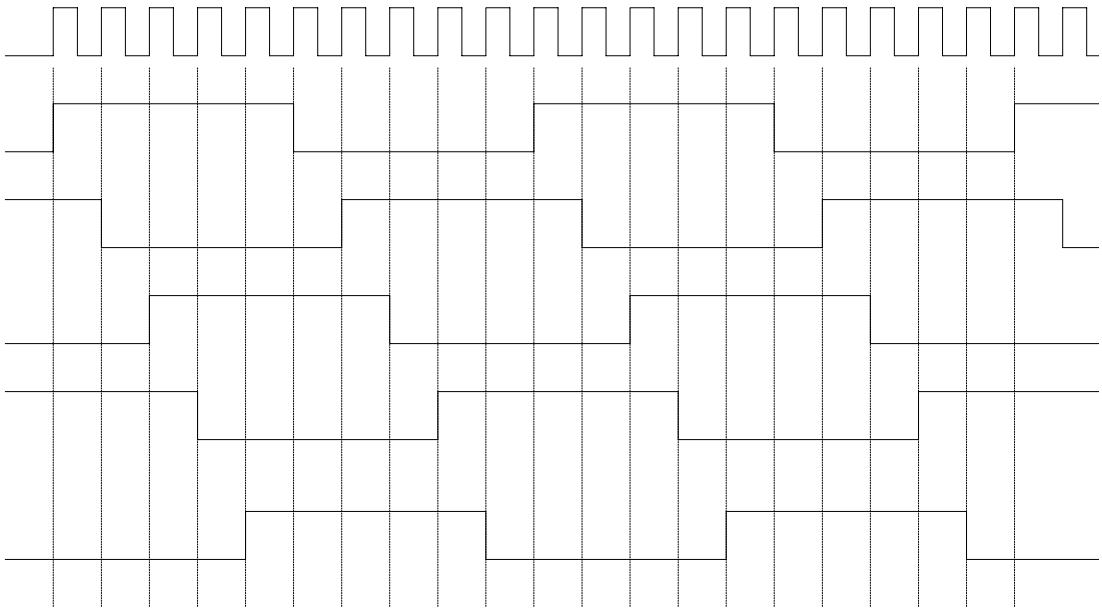


Figure 8.47. Timing diagrams for the ring oscillator of Figure 8.46

From the timing diagrams of Figure 8.47 we observe that the pulse repetition frequency of the generated waveforms is ten times the pulse repetition frequency of the clock pulse. In general, a ring oscillator with N inverters, where N is an odd number, and delay of t_p, will have a period of oscillation $2Nt_p$, and a frequency of $1/2Nt_p$. For instance, if the propagation delay of each inverter in a 5-inverter ring is $10\ ns$, the frequency of oscillation will be

$$1/2Nt_p = 1/(2 \times 5 \times 10 \times 10^{-9}) = 10\ \text{MHz}$$

* *For a discussion of electronic oscillators, please refer to Electronic Devices and Amplifier Circuits, ISBN 0-9744239-4-7.*

8.15 Summary

- A sequential circuit is a logic circuit whose output(s) is a function of its input(s) and also its internal state(s). Generally, sequential circuits have two outputs one of which is the complement of the other. Sequential circuits may or may not include logic gates.
- Flip flops, also known as bistable multivibrators, are electronic circuits with two stable outputs one of which is the complement of the other. The outputs will change only when directed by an input command.
- The 4 types of flip flops are the Set-Reset (SR) or Latch, D-type (Data or Delay), JK, and T-type (Toggle).
- An SR flip flop can be constructed with either NAND or NOR gates. If constructed with NAND gates, the condition where both inputs S and R are logic 0 simultaneously must be avoided. If constructed with NOR gates, the condition where both inputs S and R are logic 1 simultaneously must be avoided.
- A flip flop is said to be operating asynchronously when its inputs and present state alone determine the next state. However, most flip flops are provided with an input where an external clock, denoted as CP, is connected causing the flip flop to operate synchronously. Thus, when a flip flop operates synchronously with the clock pulse, the inputs instruct the flip flop what to do whereas the clock pulse tells the flip flop when to do it.
- Most flip flops are provided a set direct (SD) and a reset direct (RD) commands. These commands are also known as SET and CLR (CLEAR). These are asynchronous inputs, that is, they function independently of the clock pulse. Thus, SD and RD commands bypass the CP command.
- The D-type flip flop is a modification of a synchronous (clocked) SR flip flop. The input of a D-type flip flop appears at the output Q after the occurrence of the clock pulse, and for this reason the D-type flip flop is often called a data transfer flip flop. Data can be transferred from the input D to the output Q either during the leading edge of the clock pulse or during the trailing edge of the clock pulse. A commercially available D-type flip flop is the SN74LS74 IC device.
- The JK-type flip flop is a modification of the clocked (synchronous) SR flip flop where one of the two inputs designated as J behaves like an S (set command), and the other input K behaves like an R (reset command). Its most important characteristic is that any combination of the inputs J and K produces a valid output. The JK flip flop behaves as the SR flip flop except that it complements (toggles) the present state output whenever both inputs J and K are logic 1 simultaneously.
- The JK-type flip flop will malfunction if the clock pulse (CP) has a long time duration. This problem is eliminated when the JK flip flop is constructed in a master / slave configuration.

- The T-type (toggle) flip flop is a single input version of the basic JK flip flop, that is, the T flip flop is obtained from the basic JK flip flop by connecting the J and K inputs together where the common point at the connection of the two inputs is designated as T.
- A positive pulse is a waveform in which the normal state is logic 0 and changes to logic 1 momentarily to produce a clock pulse. A negative pulse is a waveform in which the normal state is logic 1 and changes to logic 0 momentarily to produce a clock pulse. In either case, the positive edge is the transition from 0 to 1 and the negative edge is the transition from 1 to 0.
- Edge triggering uses only the positive or negative edge of the clock pulse. Triggering occurs during the appropriate clock transition. Edge triggered flip flops are employed in applications where incoming data may be random. The SN74LS74 IC device is a positive edge triggered D-type flip flop. The SN74LS70 device is a JK flip flop which is triggered at the positive edge of the clock pulse. The SN74LS76 IC device is a JK flip flop which is triggered at the negative edge of the clock pulse. The SN74LS70 JK flip flop is preferable in applications where the incoming data is not synchronized with the clock whereas the SN74LS76 JK flip flop is more suitable for synchronous operations such as synchronous counters.
- The analysis of synchronous sequential logic circuits is facilitated by the use of state tables which consist of three vertical sections labeled present state, flip flop inputs, and next state. The present state represents the state of each flip flop before the occurrence of a clock pulse. The flip flop inputs section lists the logic levels (zeros or ones) at the flip flop inputs which are determined from the given sequential circuit. The next state section lists the states of the flip flop outputs after the clock pulse occurrence.
- The design of synchronous counters is facilitated by first constructing a state diagram and then the state table consisting of the present state, next state, and flip flop inputs sections. The present state and next state sections of the state table are easily constructed from the state diagram. The flip flops input section is derived from the present state and next state sections with the aid of the transition tables.
- A register is a group of binary storage devices such as flip flops used to store words of binary information. thus, a group of n flip flops appropriately connected forms an *n-bit* register capable of storing *n* bits of information. a register may also contain some combinational circuitry to perform certain tasks. Transfer of information from one register to another can be either synchronous transfer or asynchronous transfer.
- Binary words can be transferred from one register to another either in parallel or serial mode. In a parallel transfer all bits of the binary word are transferred simultaneously with a single transfer command. Some registers are provided with an additional command line referred to as the read command in addition to the transfer command which is often referred to as the load command.
- A shift register is a register that is capable of shifting the bits of information stored in that register either to the left or to the right. Either the input or the output of a shift register may be serial or parallel. A register with a serial input is one in which the incoming data are received one bit at a time, where the least significant bit of the incoming binary word is received first and the most significant bit (msb) is received last. After the lsb of the incoming data is stored in

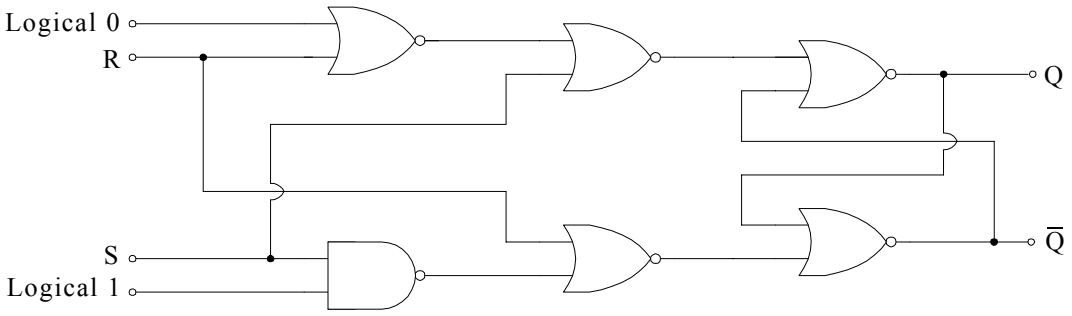
the first flip flop of the register, it is shifted to the next flip flop to the right of the first one so that the first flip flop can receive the next arriving bit.

- A shift register can be a left-shift register, a right-shift register, a serial-to-parallel register, a parallel-to-serial register, or a parallel input/parallel output register.
- Ring counters are a special type of counters that can be used to generate timing signals. In any general type ring counter, at least one flip flop of the ring counter must be preset to 1 and at least one flip flop must be reset to 0, otherwise the ring counter has no meaning since none of its outputs will change state in this case.
- The Johnson counter is a modified ring counter where the complemented output of the last flip flop is connected to the input of the first flip flop. A Johnson counter may start with all its flip flops in the reset condition, whereas a conventional ring counter requires at least one flip flop preset to logic 1 and at least one flip flop reset to logic 0. Also, a Johnson counter generates $2n$ timing signals with n flip flops whereas a conventional ring counter generates n timing signals with n flip flops.
- A ring oscillator is an arrangement of a ring consisting of an odd number of inverters, such as 3, 5, 7, and so on in a loop, and it is used in digital circuits.

8.16 Exercises

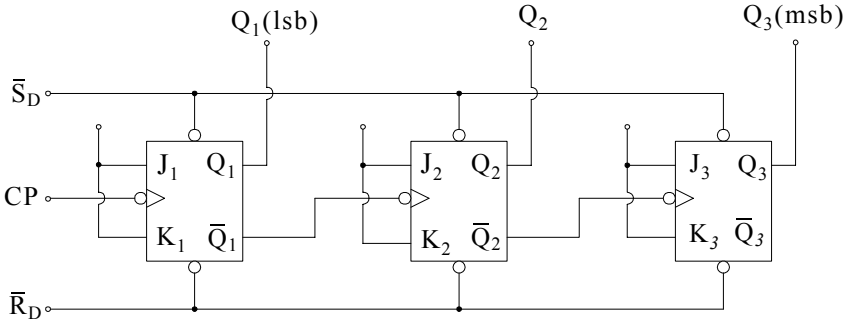
Unless otherwise stated, all JK flip flops are master/slave.

1. The logic circuit below is an asynchronous SR flip flop. By means of a characteristic table, describe its operation. How does this SR flip flop differ from the NAND-gated and NOR-gated SR flip flops?

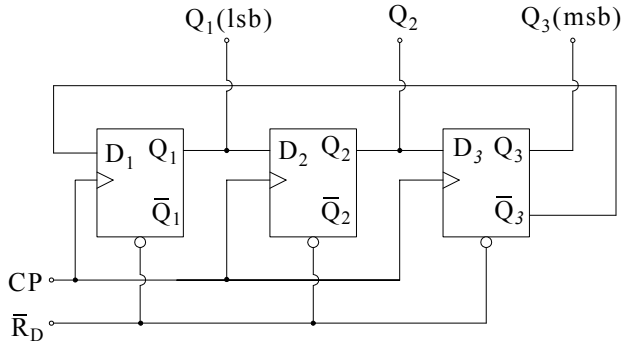


2. Modify the inputs of a master-slave JK flip flop so that it will perform like a D flip flop.

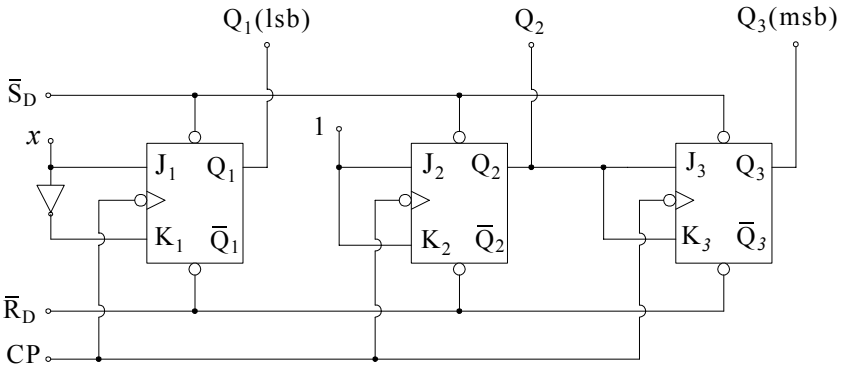
3. Describe the operation of the sequential circuit below. State any assumptions and draw a timing diagram.



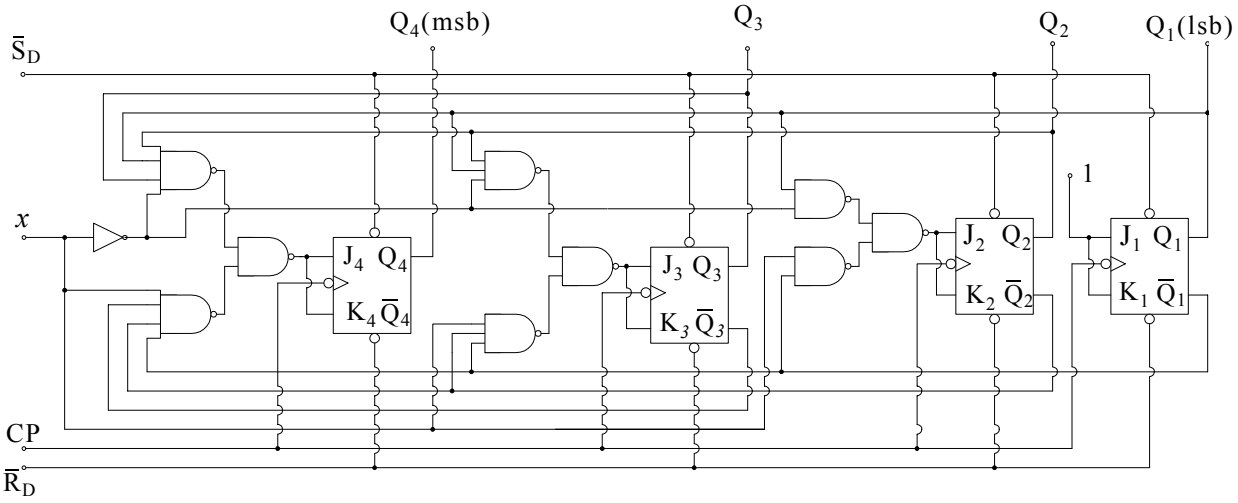
4. Describe the operation of the sequential circuit below. State any assumptions and draw a timing diagram.



5. Describe the operation of the sequential circuit below. The input x can assume the states 0 or 1. Using the state table, draw two state diagrams, one for $x=0$, and one for $x=1$.



6. Describe the operation of the sequential circuit below. The input x can assume the states 0 or 1. Using the state table, draw two state diagrams, one for $x=0$, and one for $x=1$.



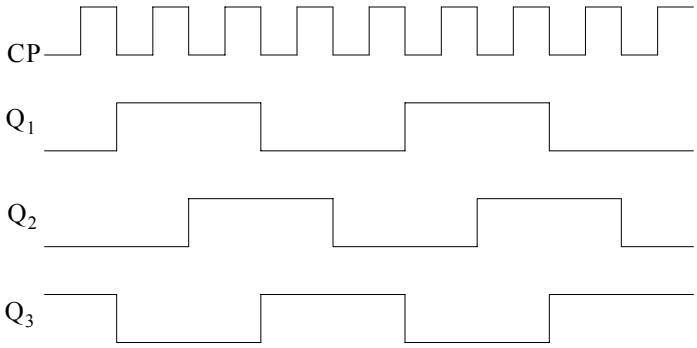
7. Design a synchronous up/down counter that will count up from zero to one to two to three, and will repeat whenever an external input x is logic 0, and will count down from three to two to one to zero, and will repeat whenever the external input x is logic 1. Implement your circuit with one TTL SN74LS76 device and one TTL SN74LS00 device.

8. Redesign the circuit of Exercise 7 above using 7474 D type flip flops and NAND gates only.

9. Design a binary counter that counts from 0 to 1 to 2, and repeats the counting sequence. Implement your circuit with one SN74LS76 JK type flip flop device.

10. Draw a block diagram for a 24-hour digital clock and explain the function of each block.

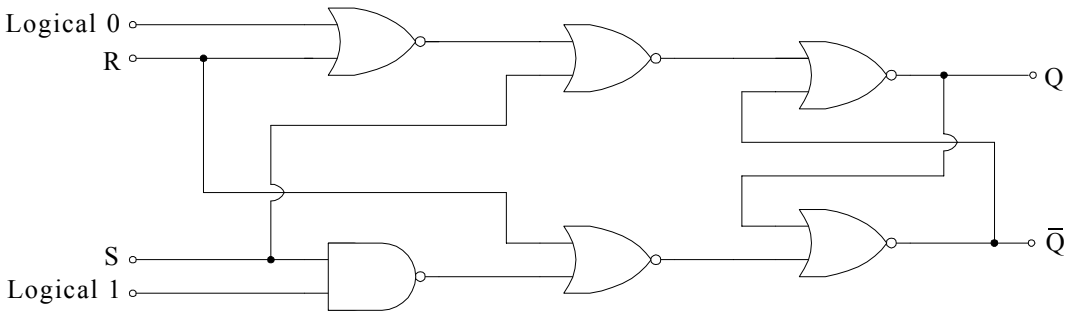
11. Design a sequential logic circuit that will produce the waveforms Q_1, Q_2, and Q_3 in accordance with the given clock pulses.



12. A ring oscillator consists of 7 inverters and it is known that the generated waveforms have a pulse repetition frequency 150 MHz. What is the propagation delay of each inverter?

8.17 Solutions to End-of-Chapter Exercises

1.



The characteristic table for this flip flop is shown below.

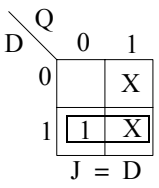
Inputs		Present State	Next State
R	S	Q_n	Q_{n+1}
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

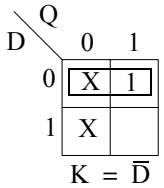
The characteristic table indicates that this flip flop tolerates simultaneous logical 0 or logical 1 inputs without causing either of the outputs to change. We recall that basic NAND-gated SR flip flop produces invalid outputs $Q = 1$ and also $\overline{Q} = 1$ when $S = R = 0$, and the basic NOR-gated SR flip flop produces the invalid outputs $Q = 0$ and $\overline{Q} = 0$ when $S = R = 1$. This problem does not exist with this logic circuit which is referred to as *exclusive OR SR flip flop*.

2. We begin with the construction of the transition table below.

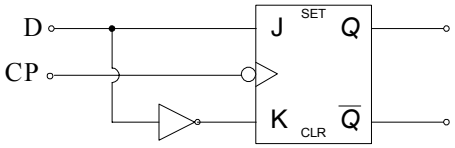
D	Q_n	Q_{n+1}	J	K
0	0	0	0	X
0	1	0	X	1
1	0	1	1	X
1	1	1	X	0

To obtain the simplest expressions for J and K in terms of D, we use the K-maps below.



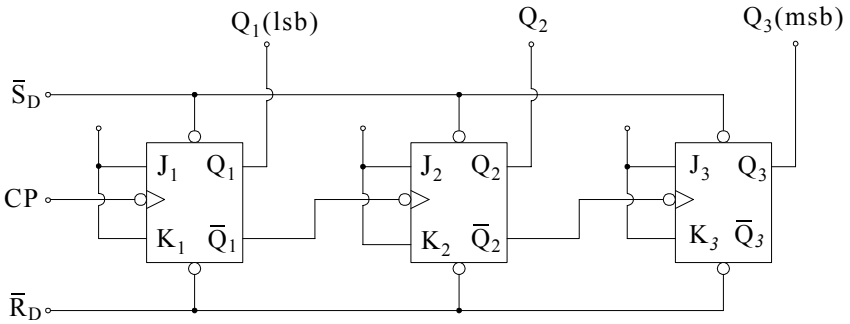


The modified flip flop is shown below.



D	Q_n	Q_{n+1}
0	0	0
0	1	0
1	0	1
1	1	1

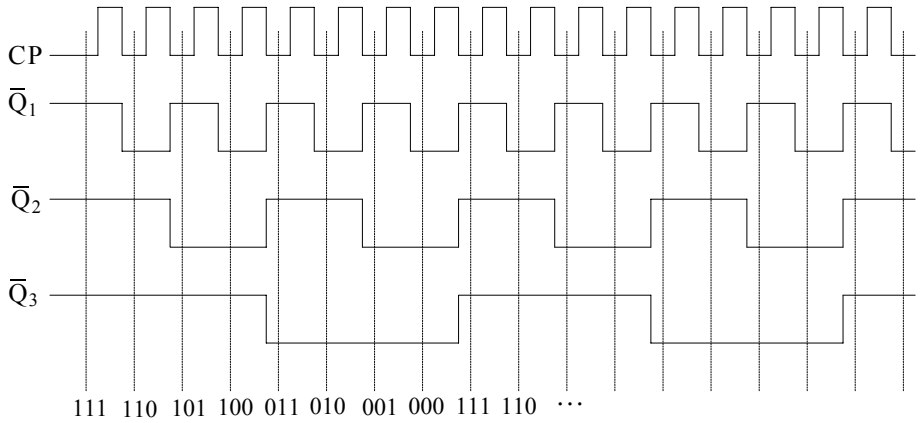
3.



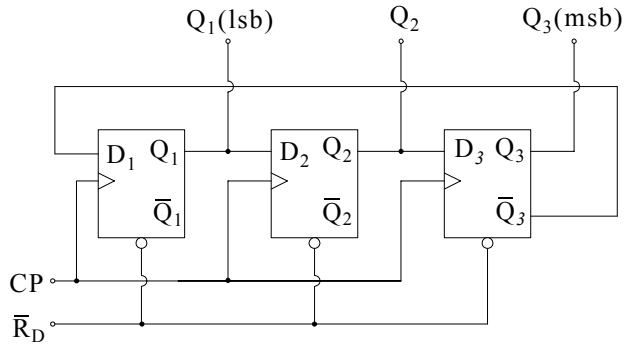
We will assume that initially all flip flops have been reset by application of the $\overline{R}_D$ command, i.e., $Q_1 = Q_2 = Q_3 = 0$. Since $J_1 = K_1 = 1$, $\overline{Q}_1$ is complemented with each clock pulse count, and since $J_2 = K_2 = 1$ also, every time $\overline{Q}_1$ changes from 1 to 0, $\overline{Q}_2$ is complemented. Likewise, flip flops J_3K_3 and J_4K_4 change state whenever Q_2 and Q_3 are complemented.

The counting sequence of the given circuit is summarized as shown in the table below. We observe that after the 8th clock pulse has occurred, the outputs of all three flip flops are set so the count is 111 (decimal 7).On the 8th clock pulse all four flip flops are reset so the count is 000 and starts a new counting cycle. The counting sequence is also obvious from the timing diagram shown in the figure below.

CP	Q_3	Q_2	Q_1	$\overline{Q}_3$	$\overline{Q}_2$	$\overline{Q}_1$
0	0	0	0	1	1	1
1	0	0	1	1	1	0
2	0	1	0	1	0	1
3	0	1	1	1	0	0
4	1	0	0	0	1	1
5	1	0	1	0	1	0
6	1	1	0	0	0	1
7	1	1	1	0	0	0
8	0	0	0	1	1	1



4.

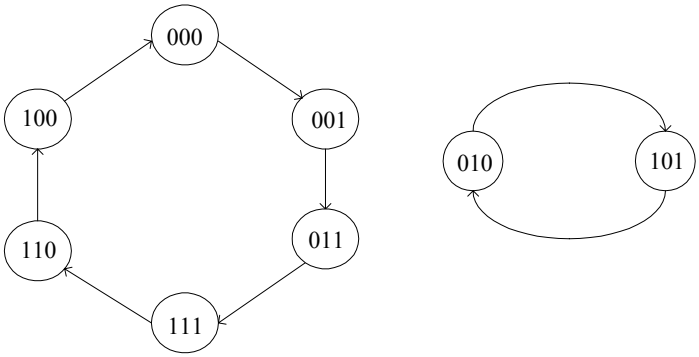


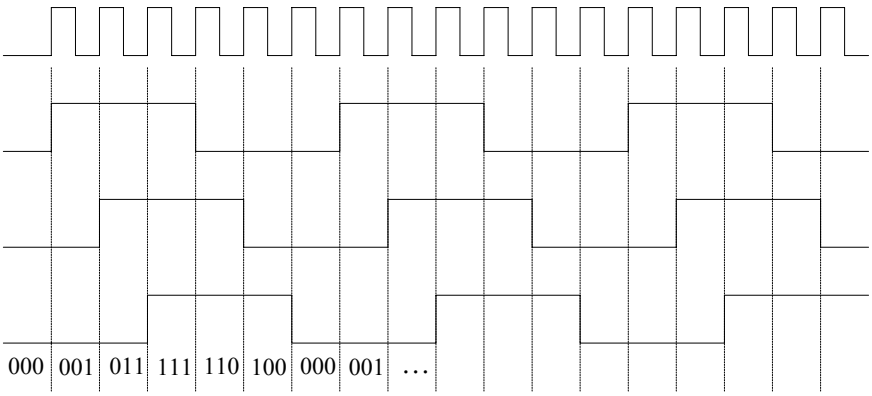
From the given circuit,

$$D_1 = \overline{Q}_3$$
$$D_2 = Q_1$$
$$D_3 = Q_2$$

From the relations above, we construct the state table, state diagram, and timing diagram shown below.

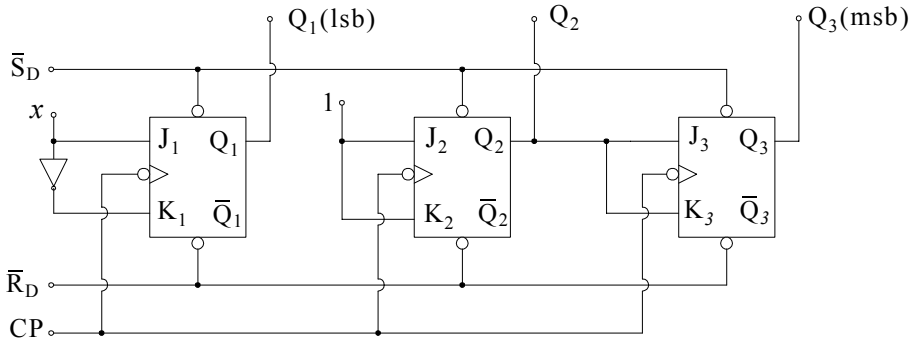
Present State			Flip Flop Inputs			Next State		
Q_3	Q_2	Q_1	D_3	D_2	D_1	Q_3	Q_2	Q_1
0	0	0	0	0	1	0	0	1
0	0	1	0	1	1	0	1	1
0	1	0	1	0	1	1	0	1
0	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0
1	0	1	0	1	0	0	1	0
1	1	0	1	0	0	1	0	0
1	1	1	1	1	0	1	1	0





Assuming that the given circuit is initially reset by the $\overline{R}_D$ command, it will perform as a *three-phase clock generator*.

5.



From the given circuit,

$$\text{For } x = 0,\ J_1 = 0,\ K_1 = 1$$

$$\text{For } x = 1,\ J_1 = 1,\ K_1 = 0$$

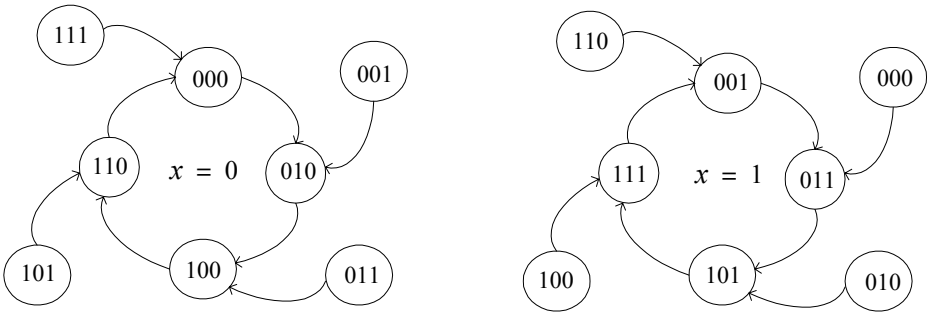
$$J_2 = K_2 = 1$$

$$J_3 = K_3 = Q_2$$

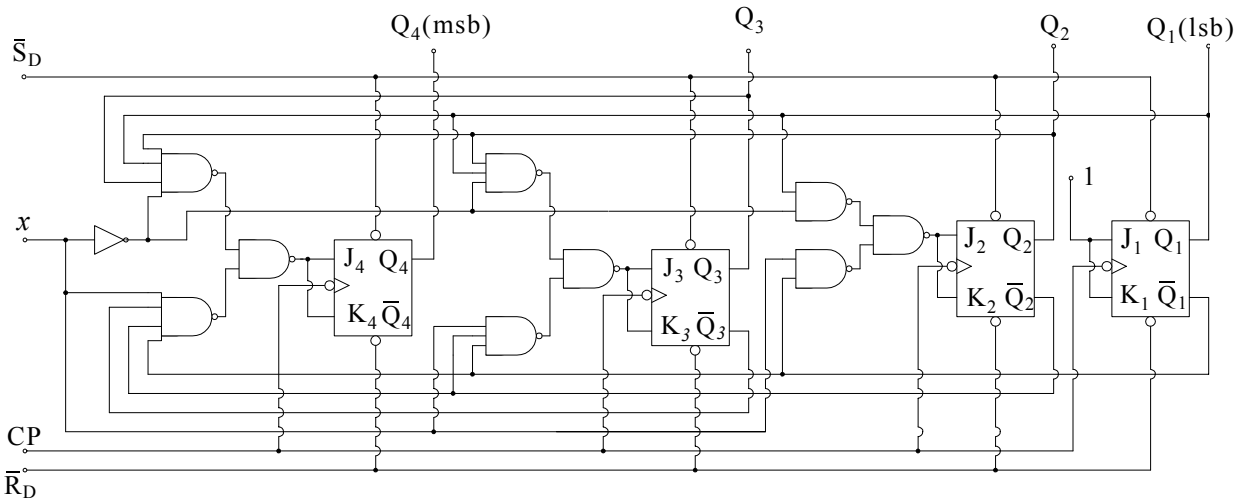
and with this information we construct the state table below.

Present State			Flip Flop Inputs												Next State					
			x=0						x=1						x=0			x=1		
Q_3	Q_2	Q_1	J_3	K_3	J_2	K_2	J_1	K_1	J_3	K_3	J_2	K_2	J_1	K_1	Q_3	Q_2	Q_1	Q_3	Q_2	Q_1
0	0	0	0	0	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1
0	0	1	0	0	1	1	0	1	0	0	1	1	1	0	0	1	0	0	0	0
0	1	0	1	1	1	1	0	1	1	1	1	1	1	0	0	1	0	0	0	1
0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1	0	0	0	1	0
1	0	0	0	0	1	1	0	1	0	0	1	1	1	0	1	0	0	0	1	1
1	0	1	0	0	1	1	0	1	0	0	1	1	1	0	1	1	0	1	0	0
1	1	0	1	1	1	1	0	1	1	1	1	1	1	0	1	1	0	1	0	1
1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0	0	1	1	0

From the state table above, we derive the state diagrams below.



6.



From the given circuit, we obtain the following expressions and we construct the state table and state diagram below.

$$J_1 = K_1 = 1$$

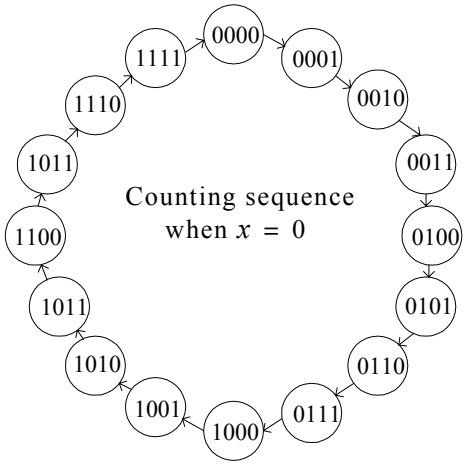
$$J_2 = K_2 = \overline{\bar{x}\bar{Q}_1} \cdot \overline{X\bar{Q}_1} = \bar{x}Q_1 + x\bar{Q}_1$$

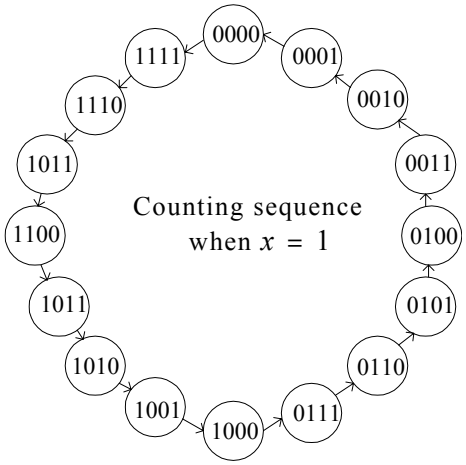
$$J_3 = K_3 = \overline{\bar{x}Q_1Q_2} \cdot \overline{x\bar{Q}_1\bar{Q}_2} = \bar{x}Q_1Q_2 + x\bar{Q}_1\bar{Q}_2$$

$$J_3 = K_3 = \overline{\bar{x}Q_1Q_2Q_3} \cdot \overline{x\bar{Q}_1\bar{Q}_2\bar{Q}_3} = \bar{x}Q_1Q_2Q_3 + x\bar{Q}_1\bar{Q}_2\bar{Q}_3$$

Present State				Flip Flop Inputs for x=0								Next State for x=0			
Q_4	Q_3	Q_2	Q_1	J_4	K_4	J_3	K_3	J_2	K_2	J_1	K_1	Q_4	Q_3	Q_2	Q_1
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1
0	0	0	1	0	0	0	0	1	1	1	1	0	0	1	0
0	0	1	0	0	0	0	0	0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1	1	1	1	1	0	1	0	0
0	1	0	0	0	0	0	0	0	0	1	1	0	1	0	1
0	1	0	1	0	0	0	0	1	1	1	1	0	1	1	0
0	1	1	0	0	0	0	0	0	0	1	1	0	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1
1	0	0	1	0	0	0	0	1	1	1	1	1	0	1	0
1	0	1	0	0	0	0	0	0	0	1	1	1	0	1	1
1	0	1	1	0	0	1	1	1	1	1	1	1	1	0	0
1	1	0	0	0	0	0	0	0	0	1	1	1	1	0	1
1	1	0	1	0	0	0	0	1	1	1	1	1	1	1	0
1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0

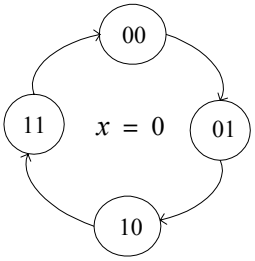
Present State				Flip Flop Inputs for x=1								Next State for x=1			
Q_4	Q_3	Q_2	Q_1	J_4	K_4	J_3	K_3	J_2	K_2	J_1	K_1	Q_4	Q_3	Q_2	Q_1
0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0
0	0	1	0	0	0	0	0	1	1	1	1	0	0	0	1
0	0	1	1	0	0	0	0	0	0	1	1	0	0	1	0
0	1	0	0	0	0	1	1	1	1	1	1	0	0	1	1
0	1	0	1	0	0	0	0	0	0	1	1	0	1	0	0
0	1	1	0	0	0	0	0	1	1	1	1	0	1	0	1
0	1	1	1	0	0	0	0	0	0	1	1	0	1	1	0
1	0	0	0	1	1	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	0	0	0	0	1	1	1	0	0	0
1	0	1	0	0	0	0	0	0	0	1	1	1	0	0	1
1	0	1	1	0	0	0	0	0	0	1	1	1	0	1	0
1	1	0	0	0	0	1	1	1	1	1	1	1	0	1	1
1	1	0	1	0	0	0	0	0	0	1	1	1	1	0	0
1	1	1	0	0	0	0	0	0	0	1	1	1	1	0	1
1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	0

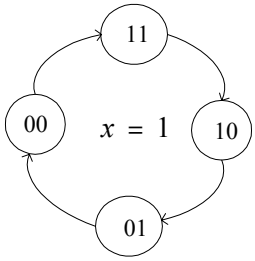




7.

First, we draw the state diagram shown below.





The SN74LS76 device is a master/slave JK flip flop and its excitation table is shown below.

JK flip flop			
Q_n	Q_{n+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Using the state diagram and the excitation table above, we derive the state table below.

Present State		Next State				Flip Flop Inputs							
		x=0		x=1		x=0				x=1			
Q_2	Q_1	Q_2	Q_1	Q_2	Q_1	J_2	K_2	J_1	K_1	J_2	K_2	J_1	K_1
0	0	0	1	1	1	0	X	1	X	1	X	1	X
0	1	1	0	0	0	1	X	X	1	0	X	X	1
1	0	1	1	0	1	X	0	1	X	X	1	1	X
1	1	0	0	1	0	X	1	X	1	X	0	X	1

We simplify the expressions for J_2 and K_2 with the K-maps below.

x \ Q_2Q_1	00	01	11	10
0		1	X	X
1	1		X	X

$$J_2 = \bar{x}Q_1 + x\bar{Q}_1$$

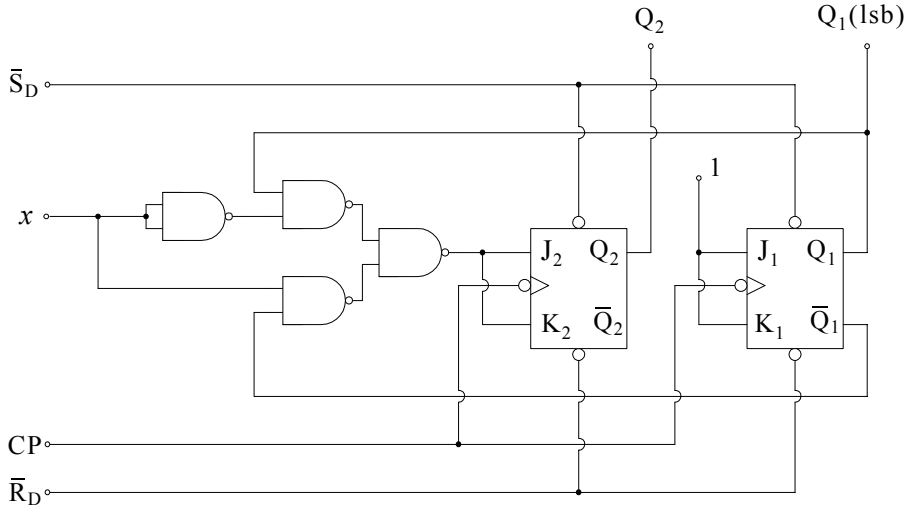
x \ Q_2Q_1	00	01	11	10
0	X	X	1	
1	X	X		1

$$K_2 = J_2 = \bar{x}Q_1 + x\bar{Q}_1$$

By inspection, from the state table above

$$J_1 = K_1 = 1$$

and the circuit is implemented with one quad SN74LS00 2-input NAND gate and one dual SN74LS76 JK flip flop as shown below.



Check:

Present State		Flip Flop Inputs								Next State			
		x=0				x=1				x=0		x=1	
Q_2	Q_1	J_2	K_2	J_1	K_1	J_2	K_2	J_1	K_1	Q_2	Q_1	Q_2	Q_1
0	0	0	0	1	1	1	1	1	1	0	1	1	1
0	1	1	1	1	1	0	0	1	1	1	0	0	0
1	0	0	0	1	1	1	1	1	1	1	1	0	1
1	1	1	1	1	1	0	0	1	1	0	0	1	0

8.

D flip flop		
Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

Using the state diagram and the excitation table above, we derive the state table below.

Present State		Next State				Flip Flop Inputs			
		x=0		x=1		x=0		x=1	
Q_2	Q_1	Q_2	Q_1	Q_2	Q_1	D_2	D_1	D_2	D_1
0	0	0	1	1	1	0	1	1	1
0	1	1	0	0	0	1	0	0	0
1	0	1	1	0	1	1	1	0	1
1	1	0	0	1	0	0	0	1	0

We attempt to simplify the expressions for D_2 and D_1 with the K-maps below.

x \ Q_2Q_1	00	01	11	10
0		1		1
1	1		1	

$D_2 = \bar{x}Q_1 + x\overline{Q}_1$

x \ Q_2Q_1	00	01	11	10
0	1			1
1	1			1

$D_1 = \overline{Q}_1$

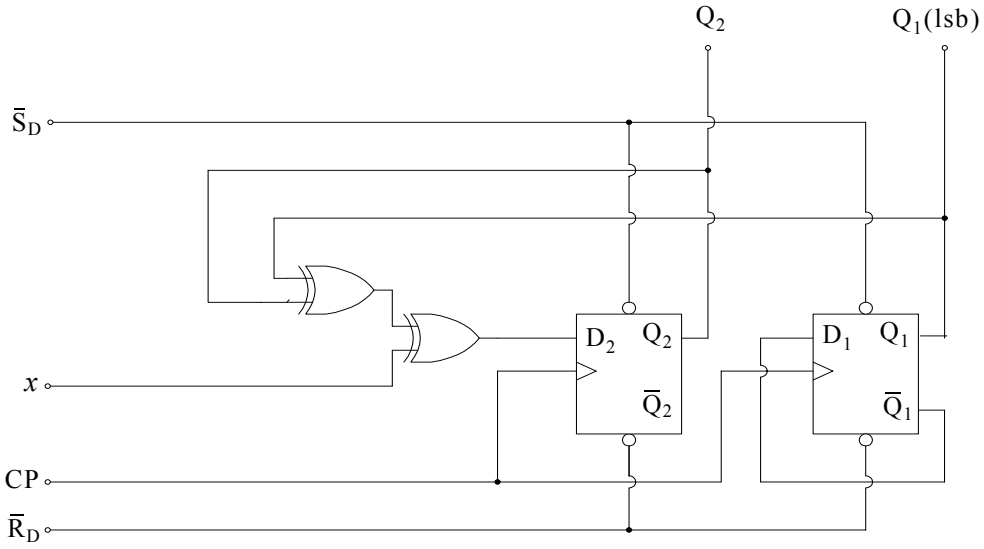
The K-map for D_1 shows that $D_1 = \overline{Q}_1$ but no simplification is possible for D_2. Thus,

$$D_1 = \overline{Q}_1$$

$$D_2 = \bar{x}\overline{Q}_2Q_1 + \bar{x}Q_2\overline{Q}_1 + x\overline{Q}_2\overline{Q}_1 + xQ_2Q_1$$

$$= \bar{x}(Q_1 \oplus Q_2) + x\overline{(Q_1 \oplus Q_2)} = x \oplus (Q_1 \oplus Q_2)$$

and we can implement this up/down counter with a dual SN74SL74 positive-edge-triggered D-type flip flop device and a quad 2-input exclusive OR gate, device SN74LS86 as shown below.

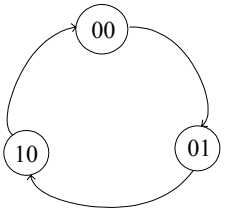


Check:

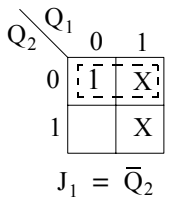
Present State		Flip Flop Inputs				Next State			
		x=0		x=1		x=0		x=1	
Q_2	Q_1	D_2	D_1	D_2	D_1	Q_2	Q_1	Q_2	Q_1
0	0	0	1	1	1	0	1	1	1
0	1	1	0	0	0	1	0	0	0
1	0	1	1	0	1	1	1	0	1
1	1	0	0	0	0	0	0	1	0

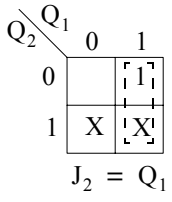
9.

The state diagram, state table, K-maps for simplification, and the implemented circuit is shown below.



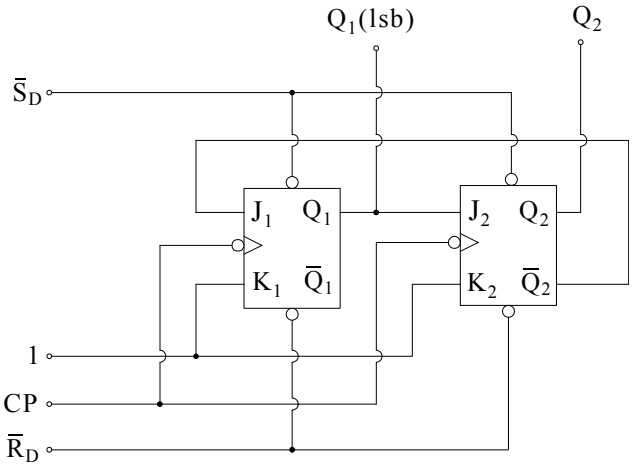
Present State		Next State		Flip Flop Inputs			
Q_2	Q_1	Q_2	Q_1	J_2	K_2	J_1	K_1
0	0	0	1	0	X	1	X
0	1	1	0	1	X	X	1
1	0	0	0	X	1	X	1
1	1	X	X	X	X	X	X





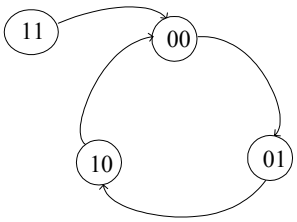
By inspection,

$$K_1 = K_2 = 1$$

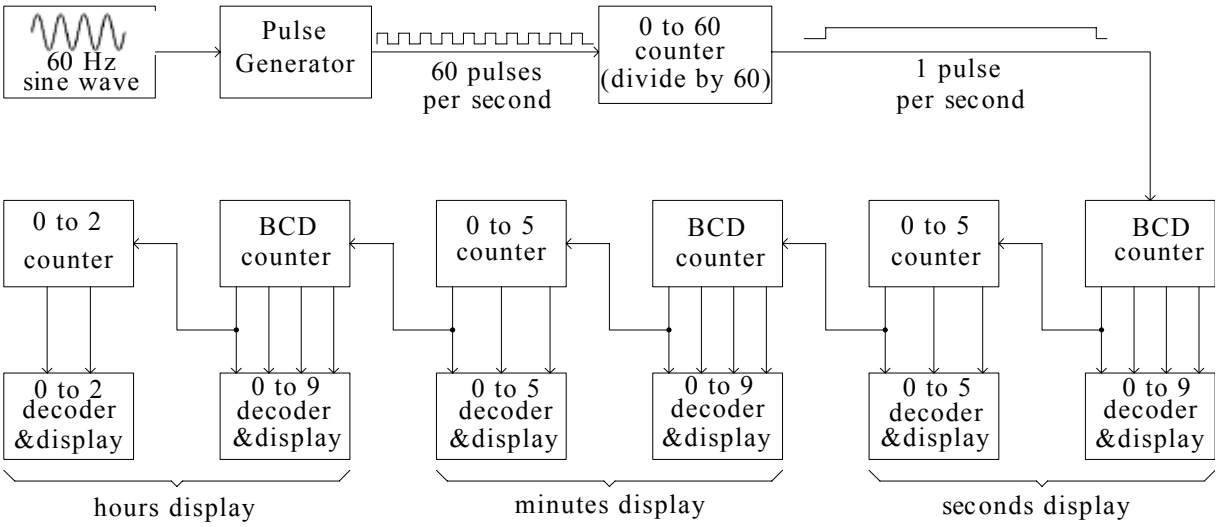


Check:

Present State		Flip Flop Inputs				Next State	
Q_2	Q_1	J_2	K_2	J_1	K_1	Q_2	Q_1
0	0	0	1	1	1	0	1
0	1	1	1	1	1	1	0
1	0	0	1	0	1	0	0
1	1	1	1	0	1	0	0

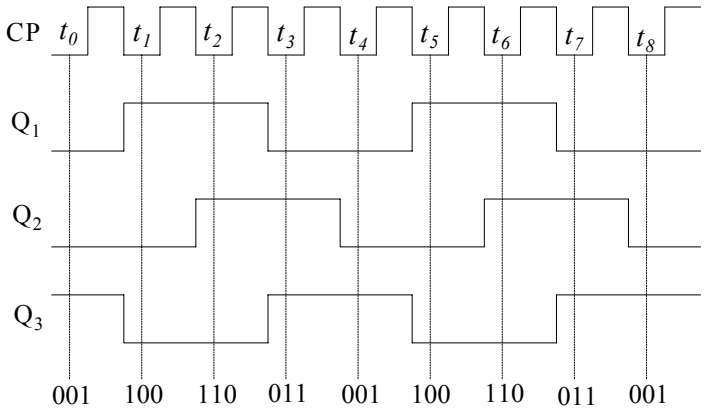


10.



First, the 60 Hz sinusoidal signal is converted to a 60 pulse per second waveform by the pulse generator and then it is divided by 60 to produce a 1 pulse per second waveform. This signal is used as a clock pulse to count and display seconds (0 to 59). The BCD counter advances one count per second and after 9 seconds the BCD counter returns to zero at which time the 0 to S counter changes from state 000 to state 001. Eventually, this counter reaches the 59 second count at which time the next clock pulse resets the 0 to 5 counter and starts the BCD counter of the minutes section, which counts at the rate of one pulse per minute. The counting sequence continues until the clock reads 59 minutes and 59 seconds at which time the next clock pulse resets the minutes and seconds and starts the BCD counter of the hours section which counts at the rate of one pulse per hour. Additional circuitry which is not shown must be provided to clear all counters before the sequence is repeated.

11.



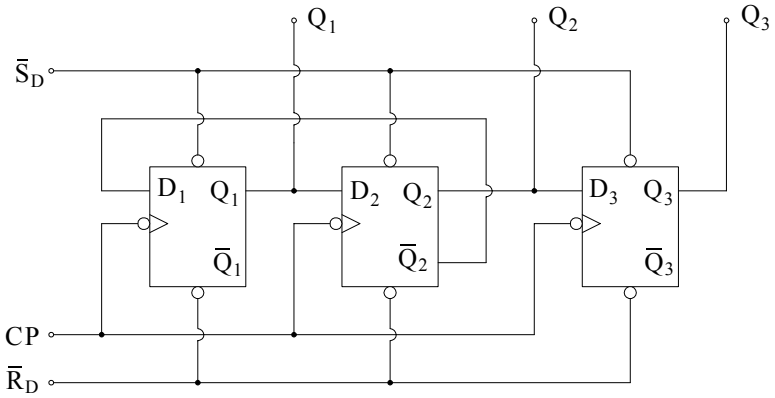
We observe that Q_3 is initially preset to logic 1 while Q_1 and Q_2 are reset to logic 0. Using the time diagram counting sequence, we construct the state table below.

Present State			Next State			Flip Flop Inputs		
Q_3	Q_2	Q_1	Q_3	Q_2	Q_1	D_3	D_2	D_1
0	0	1	1	0	0	1	0	0
1	0	0	1	1	0	1	1	0
1	1	0	0	1	1	0	1	1
0	1	1	0	0	1	0	0	1

From the state table above we observe that

$$D_1 = \overline{Q}_2$$
$$D_2 = Q_1$$
$$D_3 = Q_2$$

and from these relations, we construct the ring counter shown below.



12.

$$\text{Propagation delay} = \text{Clock pulse period} = 7 \times 150 = 1.05\text{GHz}$$

Chapter 9

Memory Devices

This chapter is an introduction to computer memory devices. We discuss the random-access memory (RAM), read-only memory (ROM), row and column decoders, memory chip organization, static RAMs (SRAMs) dynamic RAMs (DRAMs), volatile, nonvolatile, programmable ROMs (PROMs), Erasable PROMs (EPROMs), Electrically Erasable PROMs (EEPROMs), flash memories, and cache memory.

9.1 Random-Access Memory (RAM)

Random access memory (RAM) is the best known form of computer memory. RAM is considered "random access" because we can access any memory cell directly if we know the row and column that intersect at that cell. In a typical RAM, the access time is independent of the location of the data within the memory device, in other words, the time required to retrieve data from any location within the memory device is always the same and can be accessed in any order. RAM is *volatile*, that is, data are lost when power is removed.

Another type of memory is the *serial access memory* (SAM). This type of memory stores data as a series of memory cells that can only be accessed sequentially (like a cassette tape). If the data is not in the current location, each memory cell is checked until the required data is found. SAM works very well for memory buffers, where the data is normally stored in the order in which it will be used such as the buffer memory on a video card. One advantage of SAM over RAM is that the former is nonvolatile memory, that is, stored data are retained even though power is removed.

Our subsequent discussion will be restricted to RAM devices.

In a RAM device, the access time, denoted as t_a, is illustrated with the timing diagram of Figure 9.1.

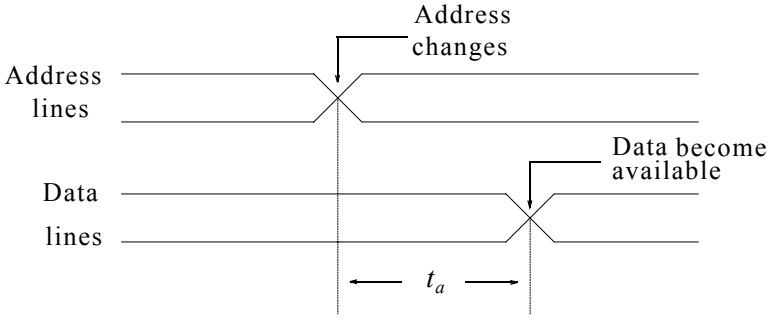


Figure 9.1. Access time defined

Figure 9.2 shows a block diagram of the input and output lines of a typical RAM device.

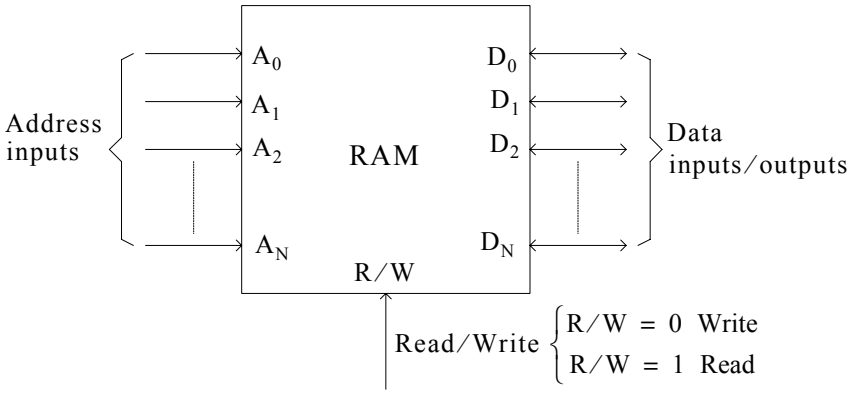


Figure 9.2. Block diagram of a typical RAM device

The bits on a memory device can be either individually addressed, or can be addressed in groups of 4, 16, 64, and so on. Thus, a 64M-bit device in which all bits are individually addressed, is known as $64M \times 1$, that is, the device is configured as a 26-bit address where $2^{26} = 67,108,864$, and for convenience it is referred to as 64M words × 1 bit memory device. The bulk part of a memory device consists of cells where the bits are stored. We may think of a cell being a D-type flip flop, and the cells are organized in a square matrix.

Figure 9.3 shows a block diagram of a $67,108,864$ (8192×8192) words by 1 bit, referred to as $64M \times 1$ RAM device arranged in a 8192 by 8192 square matrix.

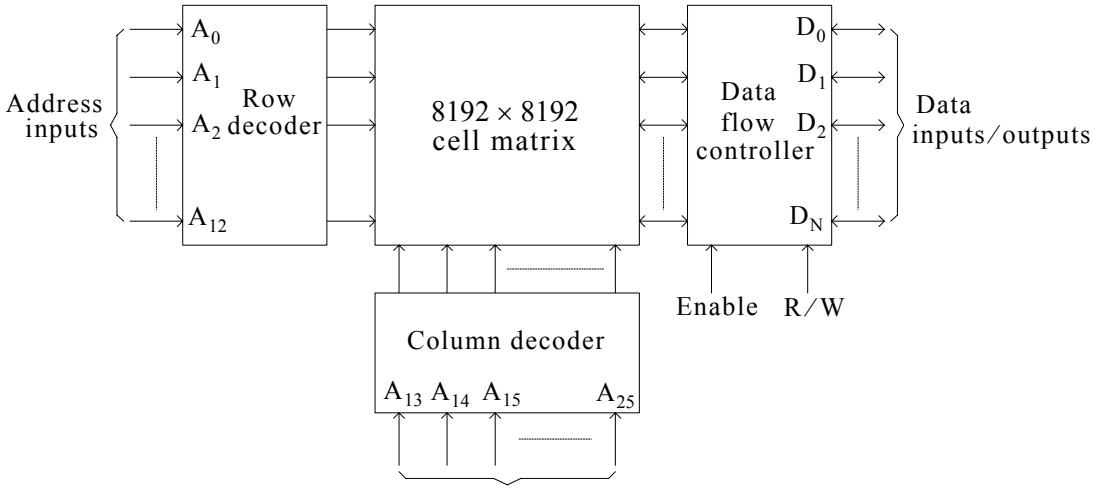


Figure 9.3. A 64M words by 1 bit RAM device

In Figure 9.3, during a read operation, a cell is selected for reading or writing by enabling its row through the row-address decoder, and its column through the column-address decoder. A sense

amplifier (not shown) detects the contents of the selected cell and provides it to the data-output terminal of the device. A similar operation is used for a write operation.

Example 9.1

We have an abundance of RAM chips each with 4096 rows and 256 columns and we want to construct a 16M bit memory.

a. How many RAM chips do we need to form this memory?

b. How many row address lines, column address lines, and lines for the RAM chips do we need for this memory?

Solution:

a. The actual size of a 16M bit memory is $16,772,216$ and thus we will need

$$\frac{16,772,216}{4096 \times 256} = 16 \text{ of } 4096 \times 256 \text{ RAM chips}$$

b. $4096 = 2^{12}$ so for 12 bits we will need 12 row lines and since $256 = 2^8$, we will need 8 column lines. We will also need one line for each RAM chip for a total of 16 RAM chip select lines.

RAMs are classified as static or dynamic. *Static RAMs*, referred to as *SRAMs*, use flip flops as the storage cells. *Dynamic RAMs*, referred to as *DRAMs*, use capacitors of the storage cells but the charge in the capacitors must be periodically refreshed. The main advantage of DRAMs over SRAMs is that for a given device area, DRAMs provide much higher storage capacity.

9.2 Read-Only Memory (ROM)

Read-Only Memories (*ROMs*) are memories in which the binary information is prewritten by special methods and the contents cannot be changed by the programmer or by any software. ROMs are also nonvolatile so that they need not be reprogrammed after power has been removed. ROMs are used to perform tasks such as code conversions, mathematical table look-up, and to control special purpose programs for computers.

As with RAMs, ROMs are arranged as N words by M bits, and when the user provides an address the ROM outputs the data of the word which was written at that address. Access time for a ROM is the time between the address input and the appearance of the resulting data word. Present day ROMs are constructed with Metal Oxide Semiconductor Field Effect Transistors (MOSFETs).* However, for simplicity, we will illustrate the operation of a typical ROM with resistors and diodes.† A simplified ROM circuit is shown in Figure 9.4.

---

* *For a detailed discussion of MOSFETs, refer to Electronic Devices and Amplifier Circuits, ISBN 0-9744239-4-7.*

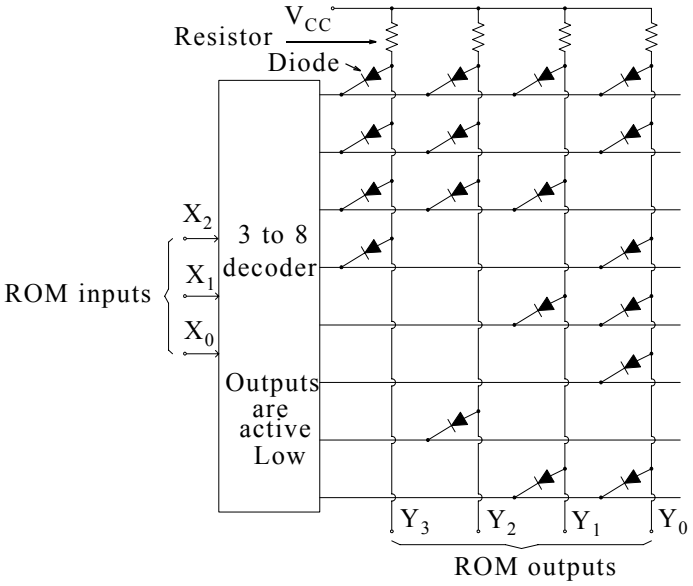


Figure 9.4. Simplified ROM circuit

The ROM circuit of Figure 9.4, consists of a decoder, a diode matrix, resistors, and a power supply. For our present discussion, a diode is an electronic device which allows electric current to flow in one direction only. A resistor is an electrical device in which when an electric current flows through, it produces a voltage drop (potential difference) across its terminals. The power supply, denoted as V_{CC}, provides power to the ROM circuit.

The input lines X_0, X_1, and X_2, select one of the 8 outputs of the decoder which when active Low, selects the appropriate word appearing at the outputs Y_0, Y_1, Y_2, and Y_3. These outputs go Low if they are connected to the decoder output lines via a diode and that the decoder output line is Low. This is illustrated with the simplified resistor-diode circuit shown in Figure 9.5.

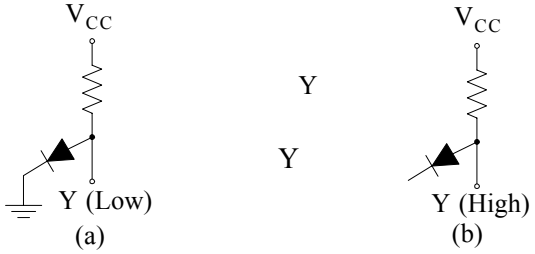


Figure 9.5. Simplified resistor-diode circuit

The output Y is Low (logic 0) whenever the selected decoder output line which is active Low allows electric current to flow through the resistor and through the diode thus developing a volt-

† *Same reference as above.*

age drop (potential difference) equal to V_{CC} as shown in Figure 9.5(a). The output Y is High (logic 1) whenever the selected decoder output line is not active Low, and in this case no current flows and thus resistor-diode forms an open circuit as shown in Figure 9.5(b).

Table 9.1 shows the outputs Y of the ROM circuit of Figure 9.4 for each input combination.

TABLE 9.1 Outputs for the ROM circuit of Figure 9.4

Inputs			Outputs			
X_2	X_1	X_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0
0	0	1	0	0	1	0
0	1	0	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	0	1	1	1	1	0
1	1	0	1	0	1	1
1	1	1	1	1	0	0

ROMs are used extensively as *look up tables* for different functions such as trigonometric functions, logarithms, square roots, etc. Table 9.2 is an example of a look up table where a 4-bit input and a 4-bit output is used to convert an angle between 0 and 90 degrees to its cosine. We recall that as $0 \le \theta \le 90°$, the cosine assumes the range of values in the interval $0 \le \theta \le 90°$

TABLE 9.2 An example of a lookup table

θ (deg)	Binary input				Output ($cos\,\theta$) - Fractional values															
0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
6	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
12	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
18	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
24	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
30	0	1	0	1	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
36	0	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
42	0	1	1	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
48	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
54	1	0	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
60	1	0	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
66	1	0	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
72	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
78	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
84	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
90	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The output values for $cos\theta$ are understood to be fractional values. For instance,

$$\cos 30° = 0.0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + \ldots + 1 \times 2^{-10} = 0.499 \approx 0.5$$

Programming instructions that are stored in a read-only memory unit rather than being implemented through software are known as *firmware*.

ROMs are also used as *character generators* referred to as CGROMs. For instance, a CGROM device may be designed to produce 128 characters in a 7 by 9 matrix. To select one of the characters, the appropriate binary code is applied at the address inputs of the device. All CGROM devices are capable of shifting the descender characters g, j, p, q, and y, below the baseline. Figure 9.6 shows the non-shifted characters A and a and the shifted character j.

9.3 Programmable Read-Only Memory (PROM)

A *programmable read-only memory* (PROM) is simply a ROM that can be programmed by hardware procedures. When first bought, PROMs contain all zeros (or all ones) in every bit of the stored binary words. The buyer can then use a PROM programmer to break certain links thus changing zeros to ones (or ones to zeros) as desired, and thus an unbroken link represents one state and a broken link another state. Figure 9.7 shows a simplified method of programming a PROM where each closed switch represents an unbroken link and an open switch represents a broken link.

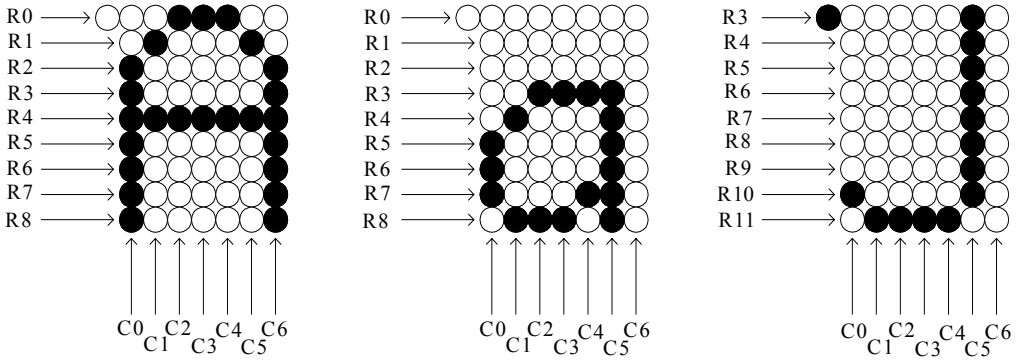


Figure 9.6. Characters in a typical CGROM

In commercially available PROMs, the open/close switches shown in Figure 9.7 are actually fuses. In a PROM, a charge sent through a column will pass through the fuse in a cell to a grounded row indicating a value of 1. Since all the cells have a fuse, the initial (blank) state of a PROM chip is all logic 0s assuming that the decoder outputs are active Low. To change the value of a cell to logic 1, we use a programmer to send a specific amount of current to the cell and this current burns the fuse. Accordingly, this process is known as burning the PROM. Obviously, PROMs can only be programmed once.

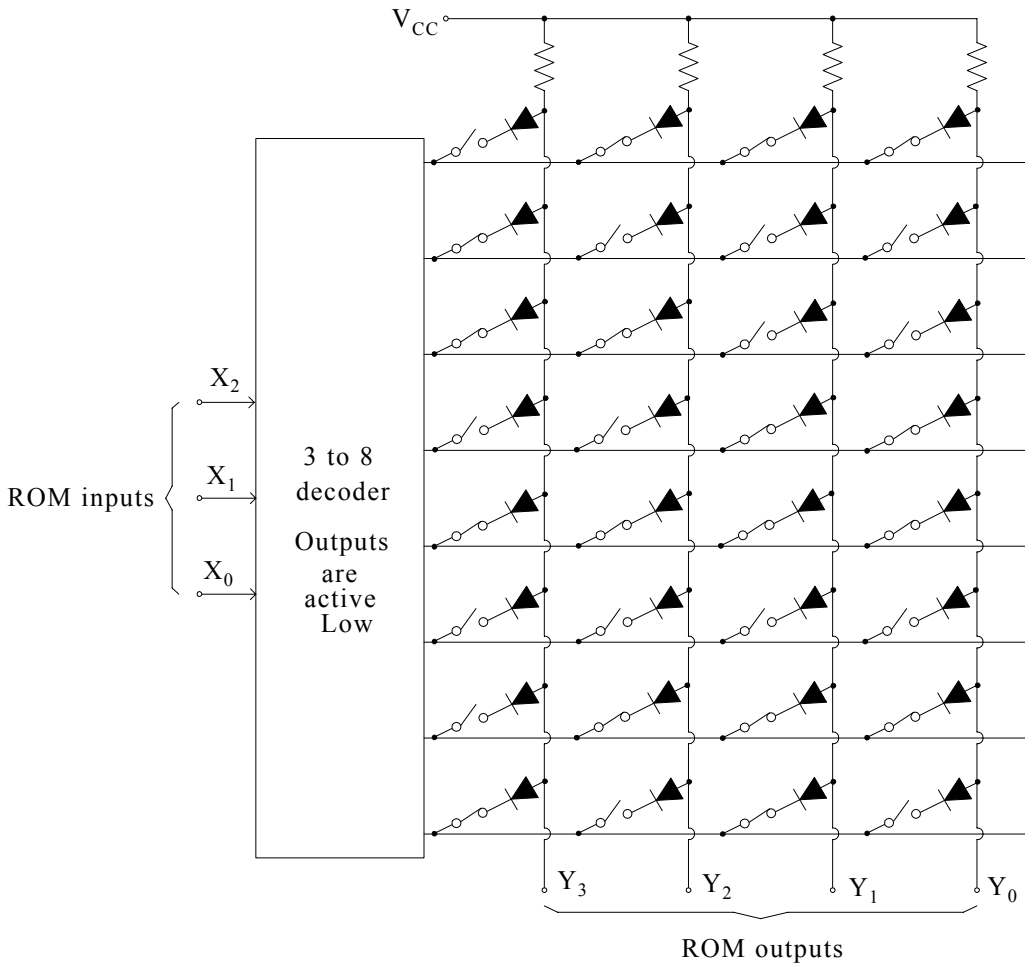


Figure 9.7. A simplified illustration of a typical PROM

9.4 Erasable Programmable Read-Only Memory (EPROM)

Erasable programmable read-only memory (*EPROM*) can be erased and reprogrammed many times. To rewrite an EPROM, we must erase it first. To erase an EPROM, a special tool that emits an ultraviolate (UV) light at a certain frequency and specified duration. In commercially available EPROMs, the cell at each intersection of a row and a column is usually an enhancement type n-channel MOSFET* with two gates, one referred to as the *floating gate,* and the other as *select (or control) gate*. These two gates are separated from each other by a thin oxide layer.

The floating gate is programmed by applying a relatively large voltage, typically 15 to 18 volts between the drain and the source of the MOSFET. This voltage causes the floating-gate to act like an electron gun. The excited electrons are pushed through and trapped on the other side of

* *For a detailed discussion on MOSFETs, please refer to Electronic Devices and Amplifier Circuits, ISBN 0-9744239-4-7.*

the thin oxide layer, creating a negative charges and these charges act as a barrier between the select gate and the floating gate. A device called a *cell sensor* monitors the level of the electron charge passing through the floating gate. If the flow through the gate is greater than 50 percent of the charge, it has a value of 1. When the charge passing through drops below the 50-percent threshold, the value changes to 0. A blank EPROM has all of the gates fully open and this results in a logic 1 in each cell of the EPROM.

9.5 Electrically-Erasable Programmable Read-Only Memory (EEPROM)

The *electrically-erasable PROM* (*EEPROM*) is a variant of the EPROM that can be erased and reprogrammed without the use of ultraviolet light. *EEPROMs* remove the biggest drawbacks of EPROMs. EEPROMs need not be removed to be rewritten, and there is no need that the entire chip has to be completely erased. Moreover, changing the contents does not require additional dedicated equipment. Generally, the contents of EEPROMs change state 1 byte at a time. However, EEPROM devices are too slow to use in many products that require quick changes to the data stored on the device.

9.6 Flash Memory

Flash memory is rewritable memory chip that holds its contents without power. It is used for easy and fast information storage in cell phones, digital cameras, and video game consoles. Flash memory works much faster than traditional EEPROMs because it writes data in chunks, usually 512 bytes in size, instead of 1 byte at a time. One of the most common uses of flash memory is for the *basic input/output system* (*BIOS*) of a computer. The task of the BIOS it to make sure that all the other devices, i.e., hard drives, ports, microprocessor, etc., function together and properly.

9.7 Memory Sticks

A *memory stick* is a device that uses flash memory. It was first introduced in 1998. A memory stick records various types of digital content, allows sharing of content among a variety of digital products, and it can be used for a broad range of applications. The dimensions of a typical memory stick is 1.97 x 0.85 x 0.11 in. A smaller size, known as *memory stick duo media*, has dimensions 1.22 x 0.79 x 0.06 in, and it is extensively used in mobile applications. The maximum theoretical access speed of a modern technology memory stick is about 150 Mbps but the actual speed depends on the design of the host device.

Generally, all memory stick media are pre-formatted by the manufacturer and they are readily available for immediate use. The manufacturer provides re-formatting instructions in the event that the user wishes to reformat the memory stick at a later date. It is also possible to transfer data from a memory stick to a PC through a memory stick USB reader/writer. The advantages of flash memory over a hard disk are that flash memory is noiseless, provides faster access, smaller size and weight, and has no moving parts. However, the big advantage of a hard disk is that the cost per megabyte for a hard disk is considerably cheaper, and the its capacity is substantially higher than that of flash memory device.

A very popular flash memory device it the SmartMedia® developed by Toshiba. SmartMedia cards are available in capacities ranging from 2 MB to 128 MB. The card is very small, approximately 45 mm long, 37 mm wide and less than 1 mm thick. SmartMedia cards can be erased, written onto, and read from memory in small blocks (256- or 512-byte increments). Accordingly, they are fast, reliable, and allows the user to select the data he wishes to keep.

9.8 Cache Memory

Cache memory is essentially a fast storage buffer in the microprocessor of a computer. Caching refers to the arrangement of the memory subsystem in a typical computer that allows us to do our computer tasks more rapidly. Thus, the main purpose of a cache is to accelerate the computer while keeping the price of the computer low. Modern computers are bundled with both L1 and L2 caches and these terms are explained below.

Let us consider the case where in a typical computer the main memory (RAM) access time is 20 nanoseconds and the cycle time of the microprocessor is 0.5 nanosecond. Without cache memory, the microprocessor is forced to operate at the RAM's access time of 20 nanoseconds. Now, let us assume that a microprocessor is build with a small amount of memory within in addition to the other components, and that memory has an access time of 05 nanosecond, the same as the microprocessor's cycle time. This memory is referred to as *Level 1 cache* or *L1 cache* for short. Let us also suppose that, in addition to L1 cache, we install memory chips on the motherboard that have an access time of 10 nanoseconds. This memory is referred to as *Level 2 cache* or *L2 cache* for short. Some microprocessors have two levels of cache built right into the chip. In this case, the motherboard cache becomes *Level 3 cache*, or *L3 cache* for short.

Cache can also be built directly on peripherals. Modern hard disks come with fast memory, hardwired to the hard disk. This memory is controlled by the hard disk controller. Thus, as far as the operating system is concerned, these memory chips are the disk itself. When the computer asks for data from the hard disk, the hard-disk controller checks into this memory before moving the mechanical parts of the hard disk (which is very slow compared to memory). If it finds the data that the computer asked for in the cache, it will return the data stored in the cache without actually accessing data on the disk itself, saving a lot of time.

9.9 Virtual Memory

Virtual memory refers to a scheme where the operating system frees up space in RAM to load a new application. Virtual memory can be thought of as an alternative form of memory caching. For instance, let us suppose that the RAM is full with several applications open at a particular time. Without virtual memory, we will get a message stating that the RAM is full and we cannot load another application unless we first close one or more already loaded applications. However, with virtual memory, the operating system looks at RAM for files that have not used recently, and copies them onto the hard disk. This creates space for another application to be loaded.

The copying from RAM to the hard disk occurs automatically, and the user is unaware of this operation. Therefore, to the user it appears that the computer has unlimited RAM space. Virtual memory provides the benefit that hard disk space can be used in lieu of a large amount of RAM

since the hard disk costs less. The area of the hard disk that stores the RAM image is called a *page file*. It holds pages of RAM on the hard disk, and the operating system moves data back and forth between the page file and RAM. On Windows platforms, page files have the .SWP extension.

Practically, all operating systems include a virtual memory manager to enable the user to configure virtual memory manually in the event that we work with applications that are speed-critical and our computer has two or more physical hard disks. Speed may be a serious concern since the read/write speed of a hard drive is much slower than RAM, and the technology of a hard drive is not designed for accessing small pieces of data at a time. If our system has to rely too heavily on virtual memory, we will notice a significant performance drop. Our computer must have sufficient RAM to handle everything we need to work with. Then, the only time we may sense the slowness of virtual memory would be when there is a slight pause when we are swapping tasks. In this case, the virtual memory allocation is set properly. Otherwise, the operating system is forced to constantly swap information back and forth between RAM and the hard disk, and this occurrence is known as *thrashing*, that is, thrashing can make our computer operate at very slow speed.

9.10 Scratch Pad Memory

A *scratch-pad memory* is a usually high-speed internal register used for temporary storage of preliminary data or notes. It is a region of reserved memory in which programs store status data. Scratch pad memory is fast SRAM memory that replaces the hardware-managed cache, and may be used to transfer variables from one task to another.

9.11 Summary

- Random access memory (RAM) is the type of memory where the access time is independent of the location of the data within the memory device, in other words, the time required to retrieve data from any location within the memory device is always the same and can be accessed in any order.
- RAM is volatile, that is, data are lost when power is removed.
- Serial access memory (SAM) is memory that stores data as a series of memory cells that can only be accessed sequentially.
- SAM is nonvolatile memory, that is, stored data are retained even though power is removed.
- The bits on a memory device can be either individually addressed, or can be addressed in groups of 4, 16, 64, and so on.
- RAMs are classified as static or dynamic. Static RAMs, referred to as SRAMs, use flip flops as the storage cells. Dynamic RAMs, referred to as DRAMs, use capacitors of the storage cells but the charge in the capacitors must be periodically refreshed. The main advantage of DRAMs over SRAMs is that for a given device area, DRAMs provide much higher storage capacity.
- Read-Only Memories (ROMs) are memories in which the binary information is prewritten by special methods and the contents cannot be changed by the programmer or by any software. ROMs are nonvolatile.
- ROMs are used extensively as *look up tables* for different functions such as trigonometric functions, logarithms, square roots, etc.
- Programming instructions that are stored in a read-only memory unit rather than being implemented through software are known as firmware.
- ROMs are also used as character generators referred to as CGROMs.
- A programmable read-only memory (PROM) is a ROM that can be programmed by hardware procedures. PROMs can only be programmed once.
- Erasable programmable read-only memory (EPROM) can be erased and reprogrammed many times. To rewrite an EPROM, we must erase it first. To erase an EPROM, a special tool that emits an ultraviolet (UV) light at a certain frequency and specified duration.
- Electrically-erasable PROM (EEPROM) is a variant of the EPROM that can be erased and reprogrammed without the use of ultraviolet light.
- Flash memory is rewritable memory chip that holds its contents without power. It is used for easy and fast information storage in cell phones, digital cameras, and video game consoles. Flash memory works much faster than traditional EEPROMs
- A memory stick is a device that uses flash memory. It records various types of digital content, allows sharing of content among a variety of digital products, and it can be used for a broad range of applications.

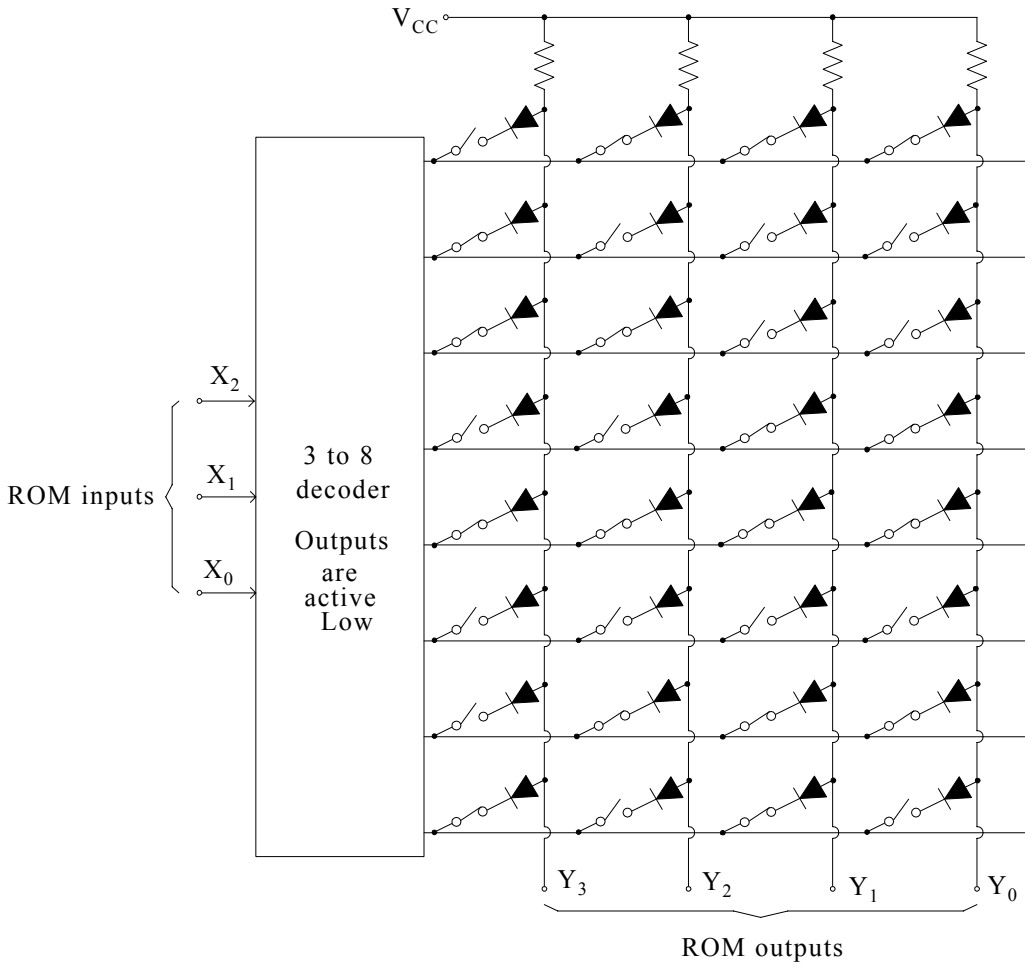
- Memory sticks are pre-formatted by the manufacturer and they are readily available for immediate use. The manufacturer provides re-formatting instructions in the event that the user wishes to reformat the memory stick at a later date. It is also possible to transfer data from a memory stick to a PC through a memory stick USB reader/writer.
- Cache memory is a fast storage buffer in the microprocessor of a computer. Caching refers to the arrangement of the memory subsystem in a typical computer that allows us to do our computer tasks more rapidly. Modern computers are bundled with both L1 and L2 caches.
- Virtual memory is a scheme where the operating system frees up space in RAM to load a new application.
- A scratch-pad memory is a usually high-speed internal register used for temporary storage of preliminary data or notes. It is fast SRAM memory that replaces the hardware-managed cache, and may be used to transfer variables from one task to another.

9.12 Exercises

1. We can expand the word size of a a RAM by combining two or more RAM chips. For instance, we can use two 32×8 memory chips where the number 32 represents the number of words and 8 represents the number of bits per word, to obtain a 32×16 RAM. In this case the number of words remains the same but the length of each word will two bytes long. Draw a block diagram to show how we can use two 16×4 memory chips to obtain a 16×8 RAM.

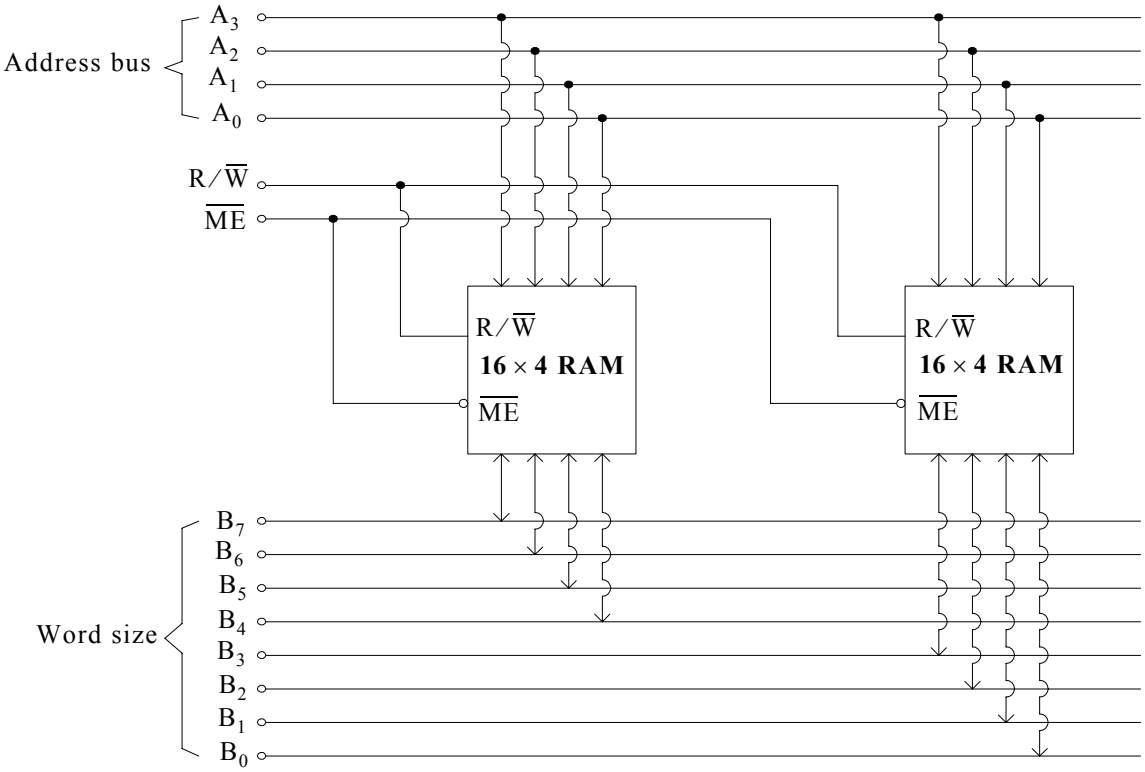
2. We can also expand the address size of a RAM by combining two or more RAM chips. Draw a block diagram to show how we can use two 16×4 memory chips to obtain a 32×4 RAM.

3. The outputs of the decoder in the ROM circuit below are active Low. List the words stored in that circuit.



9.13 Solutions to End-of-Chapter Exercises

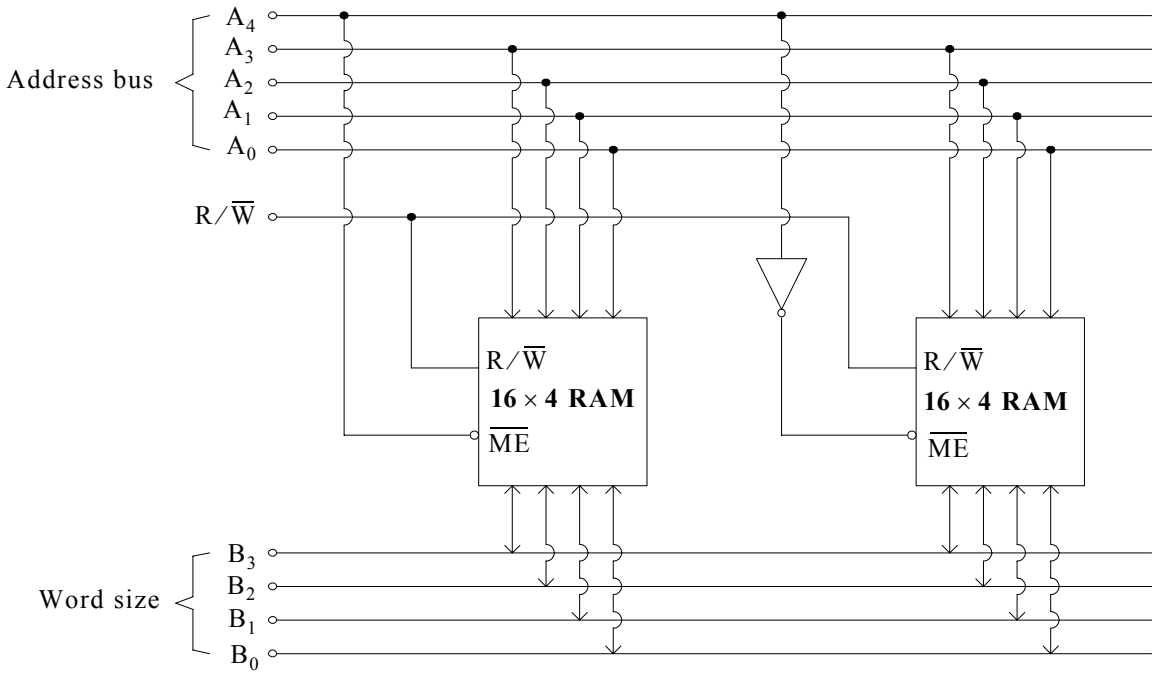
1.



The address range is from 0000 to 1111, the same as with the individual RAMs, that is, 16 words, but the word size is now 8 bits, B_0 through B_7 where the four higher order bits of the word are stored in the first 16×4 RAM and the four lower order bits of the word are stored in the second 16×4 RAM. The memory enable ($\overline{ME}$) input enables or disables the memory chip so that it will or will not respond to other inputs. Thus, when $\overline{ME}$ is logic 1, the memory chip is disabled and when $\overline{ME}$ is logic 0, the memory is enabled. The read/write ($R/\overline{W}$) input determines which memory operation is to be performed, that is, when $R/\overline{W}$ is logic 1, a read operation is performed, and when $R/\overline{W}$ is logic 0, a write operation is performed.

Similarly, we can use eight 1024×1 RAM chips to form a 1024×8 RAM.

2.



We need five address lines and since each memory chip has only four, we use the $\overline{ME}$ input as a fifth address shown as line A4 which is connected such that only one of the two RAM chips is enabled at any time. Thus, when A4 = 0, the $\overline{ME}$ of the left RAM is enabled, the $\overline{ME}$ input of the right RAM is disabled, memory locations from 00000 to 01111 are accessed. Likewise, when A4 = 1, the $\overline{ME}$ input of the left RAM is disabled, the $\overline{ME}$ input of the right RAM is enabled, and memory locations from 10000 to 11111 are accessed.

Similarly, we can use eight 1024×1 RAM chips to form an 8192×1 RAM.

3.

Row 1	1000
Row 2	0111
Row 3	0011
Row 4	1100
Row 5	0000
Row 6	1111
Row 7	1000
Row 8	0101

Chapter 10

Advanced Arithmetic and Logic Operations

This chapter begins with an introduction to the basic components of a digital computer. It continues with a discussion of the basic microprocessor operations, and concludes with the description of more advanced arithmetic and logic operations.

10.1 Computers Defined

There are two classes of computers, analog, and digital. *Analog computers* are automatic computing devices that operate with electronic signals exhibiting continuous variations of physical quantities such as electrical voltages and currents, mechanical shaft rotations or displacements, and are used primarily to solve the differential equations that describe these time-varying signals. Results are normally displayed on oscilloscopes, spectrum analyzers, and pen recorders. A basic component in analog computers is a very versatile electronic device known as operational amplifier* or op amp for short.

Digital computers are automatic computing devices that operate with electronic signals exhibiting discrete variations, and are used for a variety of tasks such as word processing, arithmetic and logic operations, database construction, e-mail, etc. Digital computer operation is based on the binary numbering system which, as we learned in Chapter 1, employs two numbers only, zero (0) and one (1). Our subsequent discussion will be on digital computers. Two important characteristics of any digital computer are the ability to store information, and the speed of operation.

Digital computers are classified either as *general purpose* (*stored program*) or *special purpose* (*dedicated*) computer. A general purpose computer is one in which the sequence of instructions (program) is read into the computer via the input unit. A special purpose computer is one in which the sequence of operations are predetermined by the actual construction of the control circuitry.

Some important digital computer terminology

Computer Interfacing is the interconnection and synchronization of digital information transmission between the main computer and separate units such as the input/output devices often referred to as *peripherals*.

Computer Hardware are the electronic and mechanical parts and materials from which the computer is fabricated.

---

* *For a thorough discussion on operational amplifiers, please refer to Electronic Devices and Amplifier Circuits, ISBN 0-9744239-4-7.*

Computer Software are programs used by a computer.

Subroutines are programs stored in memory for use by other programs. For instance, most computers have subroutines for finding trigonometric functions, logarithms, squares, square roots, e.t.c. The subroutines are normally part of the software supplied by the computer manufacturer.

Operating System is a set of programs used to control the running of the user or applications programs.

10.2 Basic Digital Computer System Organization and Operation

A typical digital computer contains the following five essential units which are interconnected as shown in Figure 10.1.

- Arithmetic/Logic Unit (ALU)
- Memory Unit
- Control Unit
- Input Unit
- Output Unit

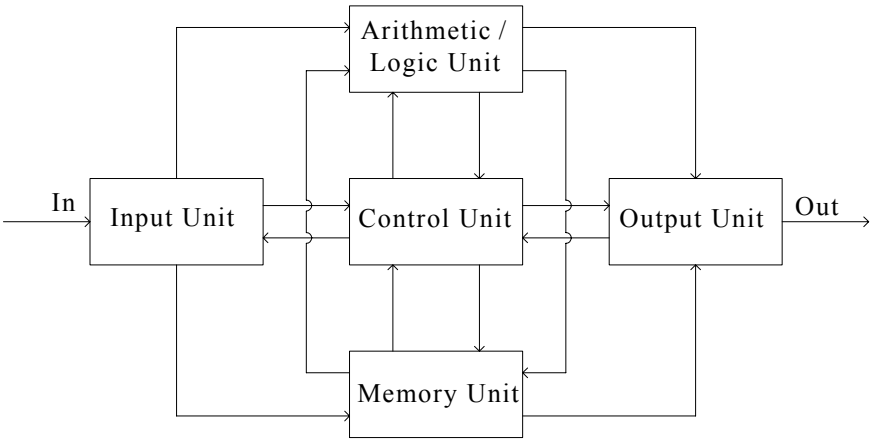


Figure 10.1. Block diagram of a basic digital computer system

The *Input Unit* consists of devices capable of taking information and data that are external to the computer and convert these inputs to a language the computer can understand. An essential function of the input unit is *analog-to-digital conversion*,* and input devices include keyboards, switches, punched cards, punched paper tape readers, and magnetic tape readers. Information

* *Analog-to-digital and digital-to-analog converters are discussed in detail in Electronic Devices and Amplifier Circuits, ISBN 0-9744239-4-7.*

from the input unit can then be sent into the memory unit or the Arithmetic / Logic Unit (ALU); this determination is made by the control unit. The input unit enters computer programs that include instructions and data in the memory unit before starting the computer operation. The input unit is also used to initiate interrupt commands such as *interrupt*, *halt*, and *pause*.

The *Output Unit* consists of devices used to transfer information and data from the computer to the outside world, i.e., the computer user. The output devices are directed by the control unit where information and data can be received either from the memory unit or from the ALU. An essential function of the output unit is *digital-to-analog conversion*. Common output devices include light emitting diodes (LEDs), printers, and monitors.

The *Arithmetic / Logic Unit* (ALU) perforce arithmetic and logic operations. The type of operation to be performed is determined by the control unit. Data that are to be operated by the ALU can be received by either the memory or the input unit. The results of the operations performed by the ALU cay be transferred to either the memory or the output unit as determined by the control unit.

The *Memory Unit* stores groups of words (binary digits) that represent instructions that the computer is to perform, and data that are to be operated on. The memory serves also as a storage for results of arithmetic/logic operations. The memory is controlled by the control unit which calls for either a *read* or *write* command. A given location in memory is accessed by the appropriate address provided by the control unit. Information can be written into memory from either the ALU or the input unit, as dictated by the control unit. The memory can also furnish data to either the ALU or the output unit, depending on the command received from the control unit.

The *Control Unit* controls the operation of the other four units (ALU, Memory, Input, and Output). The control unit consists of logic and timing circuits that generate the proper signals necessary to execute each instruction of a computer program. While the "stored program" directs the flow of operations to be performed, the control unit provides all the necessary steps that must be taken for each command to be executed. Although the control unit cay vary considerably from one computer to another, some circuitry such as decoding gates, counters, registers, and clocks are common in all control units. The control unit and the ALU are usually combined into one unit called *Central Processing Unit* (*CPU*). When a CPU is contained in a single large scale integration (LSI) chip, it is called a *microprocessor*. A typical microprocessor contains some memory.

A *microcomputer* can be thought of as a small digital computer consisting of a microprocessor and other integrated circuits (ICs) comprising the memory, input, and output units.

Every desktop and laptop computer in use today contains a microprocessor as its central processing unit. The microprocessor is the hardware component. To get its work done, the microprocessor executes a set of instructions known as software. The operating system is software that provides a set of services for the applications running on our computer, and it also provides the fundamental user interface for your computer. Examples of operating systems are Unix, Windows, and Linux. *Applications* are software that are programmed to perform specific tasks. Typical applications are word processors, spreadsheets, databases, internet browsers, and e-mail.

The *BIOS*, acronym for *basic input/output system*, is a set of essential software routines that test

hardware at startup, start the operating system, and support the transfer of data among hardware devices. The BIOS is stored in read-only memory (ROM) so that it can be executed when the computer is turned on. Present day computers use flash memory, a type of ROM as we've learned in Chapter 9.

An *integrated circuit*, commonly referred to as *chip*, is a small, thin piece of silicon onto which the transistors making up the microprocessor have been etched. A chip might be as large as an inch on a side and can contain tens of millions of transistors. Simpler processors might consist of a few thousand transistors etched onto a chip just a few millimeters square.

In our subsequent discussion we will examine some of the logic circuits used by the arithmetic / logic unit to perform certain operations.

10.3 Parallel Adder

A *parallel adder* is used to add two binary words where each word consists of several bits. Normally, the augend is stored in the *accumulator register*, which is also referred to as the A Register. The addend is stored in another register.

Example 10.1

Draw a block diagram of an N-bit parallel adder to show how the contents of two registers can be added.

Solution:

A block diagram of an N-bit parallel adder which consists of several full adders, denoted as FA, and Registers A and B is shown in Figure 10.2.

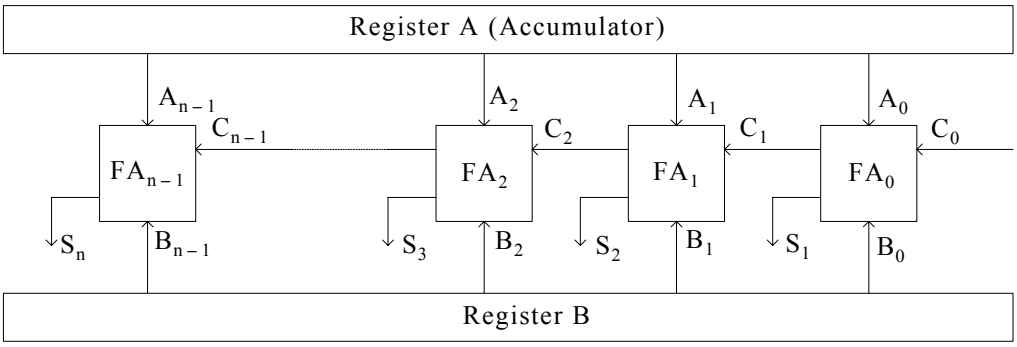


Figure 10.2. A typical N-bit parallel adder for Example 10.1

The least significant bits of each binary word are denoted as A_0 and B_0, and C_0 is the carry from a previous addition. Likewise, A_{n-1} and B_{n-1} denote the most significant bits in Registers A and B. We observe that with this arrangement each Full Adder (FA) adds the corresponding bits from registers A and B and any previous carry from the preceding FA. For instance, FA_2 adds bit

A_2 from Register A with bit B_2 from Register B and the previous carry C_2 resulting from the addition in FA_1.

A major disadvantage of the parallel adder in Figure 10.2 is the propagation delay of the carry bit from one full adder to the next higher position full adder. Accordingly, sufficient time must be allowed so that the carry bit produced by the adder of the least significant bits will be able to propagate through that adder and be available at the next higher position full adder before the addition is performed. Consider, for instance, a 6-stage (6 full adders) parallel adder in which each full adder has a propagation delay of 20 nanoseconds. In this case, the full adder immediately to the left of the least significant position full adder must wait 20 nanoseconds before it adds bits A_1, B_1, and C_1. Likewise, the next full adder must wait 40 nanoseconds after the start of the addition, the next full adder must wait 60 nanoseconds, and so on. Obviously, the situation becomes worst as more full adders are added to the circuit. This problem is alleviated by a scheme known as *look-ahead-carry* which will be discussed in a subsequent section of this chapter.

10.4 Serial Adder

The *serial adder* employs only one full adder and the binary words are added one bit at a time.

Example 10.2

Draw a block diagram of a 4-bit serial adder to add the contents of two registers.

Solution:

A block diagram of a typical 4-bit serial adder, where the upper four flip flops represent Register A and the lower four flip flops represent Register B is shown in Figure 10.3.

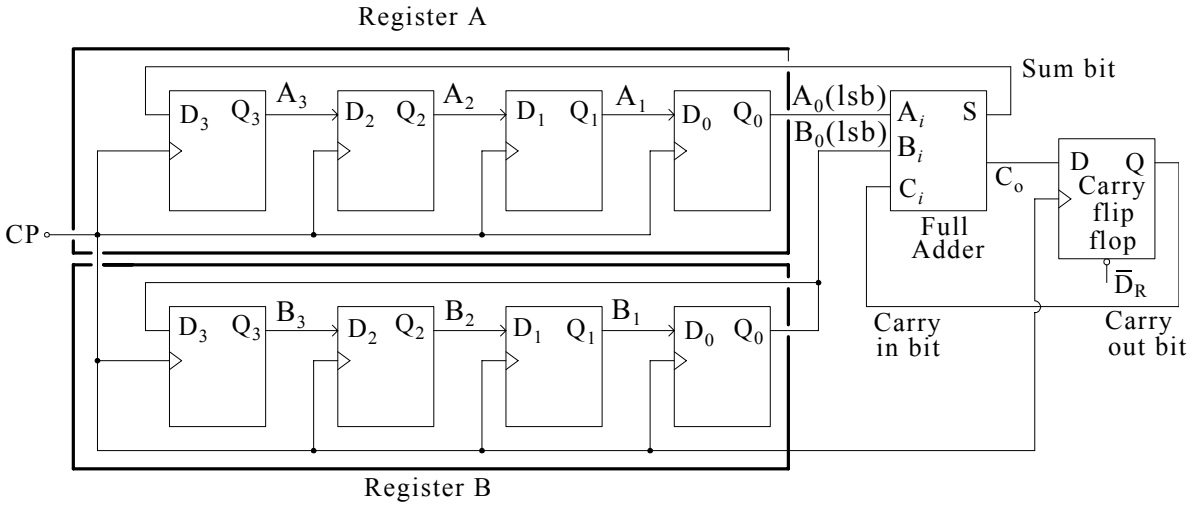


Figure 10.3. A typical 4-bit serial adder for Example 10.3

In Figure 10.3, both Registers A and B are shift registers and shift the data from left to right. The flip flop on the right is designated as a *carry flip flop* because is used to store the carry output C_o so that it will be added to the next significant bit position. We observe that after one clock pulse, C_o becomes the carry-in bit C_i. We note also that the *sum* output of the full adder is transferred and stored into the msb position of Register A, then is shifted to the right, that is, the result (sum) of the addition will be stored in the A register. Obviously, the initial contents (augend) of the A register are lost but the contents of B register are preserved since after four clock pulses the addend data have been restored into B register.

The serial adder has fewer components but it is much slower than a parallel adder. Either a serial or a parallel adder can also perform the operation of subtraction. This is because, as we have seen, two binary numbers can be subtracted by taking either the 2s complement, or the ls complement of the subtrahend and thus converting the subtraction operation into an addition operation. We recall that the 2s complement of a binary number can be obtained from the ls complement by adding a 1 to the lsb position of the ls complement of a number. Also, the ls complement of a binary number is formed by complementing (inverting) every bit of the given binary number. The design of a 4-bit parallel adder/subtractor is left as an exercise for the reader at the end of this chapter.

An adder/subtractor circuit must be capable of adding or subtracting negative as well as positive numbers. Usually, in an 8-bit (one byte) binary word, we use the msb as the sign (+ or –) bit. Thus, if the msb is 0, the number is understood to be positive and in its true form. For example the binary word 01011001 is a positive number equivalent to decimal +89. On the other hand, if the msb is 1, the number is negative and it is in its 2s complement form. For instance, the binary word 10100110 is a negative number equivalent to decimal –90.

10.5 Overflow Conditions

An *overflow* is a condition in which a calculation produces a unit of data too large to be stored in the location allotted to it. An overflow cannot occur when two numbers of opposite sign are added. This is because the addition of a positive with a negative number produces a sum (positive or negative) whose numerical value is less than the numerical value of the larger of the added numbers. This is illustrated in Examples 10.3 and 10.4 which follow and it is assumed that the sum is to be stored in a 4-bit register, the positive numbers are in their true form and the negative numbers are in the 2s complement form.

Example 10.3

Add the numbers –3 and +6 in binary form.

Solution:

$$\begin{array}{l} (-3) \rightarrow 1101 \\ (+6) \rightarrow 0110 \\ (+3) \rightarrow 0011 \end{array} \; +$$

The carry out of the msb position addition is lost since the sum has been stored in a 4-bit register.

Example 10.4

Add the numbers –6 and +5 in binary form.

Solution:

$$\begin{array}{l} (-6) \rightarrow 1010 \\ (+5) \rightarrow 0101 \\ (-1) \rightarrow 1111 \end{array} \; +$$

An overflow may occur in an addition if the augend and the addend are both positive or both negative as illustrated with Examples 10.5 and 10.6.

Example 10.5

Add the numbers +6 and +7 in binary form.

Solution:

$$\begin{array}{l} (+6) \rightarrow 0110 \\ (+7) \rightarrow 0111 \\ (+13) \rightarrow 01101 \end{array} \; +$$

We observe that the sum (+13) exceeds the register capacity because a 4-bit register has a maximum capacity of +7 and –8. Therefore, an overflow condition has occurred.

Example 10.6

Add the numbers –6 and –7 in binary form.

Solution:

$$\begin{array}{l} (-6) \rightarrow 1010 \\ (-7) \rightarrow 1001 \\ (-13) \rightarrow 10011 \end{array} \; +$$

We observe that the sum –13 exceeds the register capacity because a 4-bit register has a maximum capacity of +7 and –8. Therefore, an overflow condition has occurred.

An overflow condition can also be detected by examining the sign bit position of the sum and and the carry from the addition of the sign bit positions of the augend and the addend. If these two bits are different, then an overflow condition has occurred. This is seen in Examples 10.5 and 10.6 above, and also in examples 10.7 and 10.8 which follow where 8-bit registers are used to store the sum.

Example 10.7

Add the numbers +95 and +56 in binary form.

Solution:

$$\begin{aligned} (+95) &\rightarrow \ 01011111 \\ (+56) &\rightarrow \ 00111000 \ + \\ (+151) &\rightarrow 010010111 \end{aligned}$$

The sum +151 exceeds the register capacity since an 8-bit register has a maximum capacity of +127 and minimum of –128. Thus, an overflow occurs and this can also be detected by examining the sign bit of the sum and the carry from the addition of the sign bit positions of the augend and the addend.

Example 10.8

Add the numbers –86 and –69 in binary form.

Solution:

$$\begin{aligned} (-86) &\rightarrow \ 10101010 \\ (-69) &\rightarrow \ 10111011 \ + \\ (-155) &\rightarrow \ 101100101 \end{aligned}$$

The sum +151 exceeds the register capacity since an 8-bit register has a maximum capacity of +127 and minimum of –128. Thus, an overflow occurs and this can also be detected by examining the sign bit of the sum and the carry from the addition of the sign bit positions of the augend and the addend.

We can implement a logic circuit which detects an overflow condition by using an XOR gate with inputs the sign bit of the sum and the carry from the addition of the sign bit positions of the

augend and the addend. Thus, an overflow condition can be detected when the output of the XOR is logic 1.

10.6 High-Speed Addition and Subtraction

We found out earlier that a major disadvantage of the parallel adder circuit is that the propagation delay of each full adder must be considered before the next higher bit position is performed. Since each full adder circuit consists of two gate levels, the total propagation delay of a parallel adder circuit with N full adders will be $2N \times t_{pd}$ where t_{pd} is the propagation delay of each gate. This propagation delay problem is alleviated by a scheme known as *look-ahead carry* which adds directly the augend A_i with the addend B_i of the *ith* position full adder in the parallel adder circuit and the lsb position carry C_0. To see how this is done, let us consider the block diagram of the *ith* full adder of an N-stage parallel adder circuit shown in Figure 10.4.

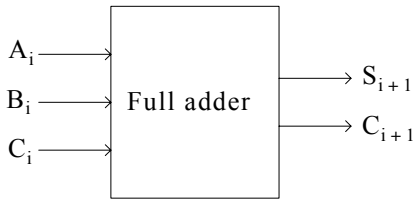


Figure 10.4. Block diagram of the ith full adder in an n-stage parallel adder circuit

In Chapter 7 we found that a carry output C_{i+1} is generated whenever the logical expression (10.1) below is true.

$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i = A_iB_i + (A_i + B_i)C_i \tag{10.1}$$

Next, we let

$$\begin{aligned} G_i &= A_iB_i \\ P_i &= A_i + B_i \end{aligned} \tag{10.2}$$

Substitution of (10.2) into (10.1) yields

$$C_{i+1} = G_i + P_iC_i \tag{10.3}$$

The term G_i represents a *generated carry* from the *ith* full adder and thus if $G_i = 1$, then C_{i+1} is independent of C_i. * The term P_iC_i represents a *propagated carry* because whenever the condition $P_iC_i = 1$ exists, then C_i will propagate to C_{i+1}.

Now, we observe that relation (10.3) contains C_i which is not a direct input to the adder circuit – the direct input† to the adder circuit is C_0 – and therefore we need to expand relation (10.3) until

* *We recall from Chapter 5 that* $1 + A = 1$

† *By direct input we mean the external carry into the parallel binary adder shown as* C_0 *in Figure 10.2*

it contains only the direct carry input C_0. This can be done by repeated substitutions of expression (10.3) above for decreasing values of the index *i* as follows:

By comparison with (10.3),

$$C_i = G_{i-1} + P_{i-1}C_{i-1} \tag{10.4}$$

Then,

$$\begin{aligned} C_{i+1} &= G_i + P_iC_i = G_i + P_i(G_{i-1} + P_{i-1}C_{i-1}) = G_i + P_iG_{i-1} + P_iG_{i-1} + P_iP_{i-1}C_{i-1} \\ &= G_i + P_iG_{i-1} + P_iP_{i-1}(G_{i-2} + P_{i-2}C_{i-2}) \\ &= G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + P_iP_{i-1}P_{i-2}C_{i-2} \\ &\quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\ &= G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \ldots + P_iP_{i-1}P_{i-2}\ldots P_1P_0C_0 \end{aligned} \tag{10.5}$$

The last expression in (10.5) reveals that each carry C_{i+1} can be implemented with only three gate delays instead of *2N* delays for a parallel adder such as that shown in Figure 10.2. The first gate delay is caused by the implementation of G_i and P_i, the second gate delay is caused by the ANDing of the gates required to form each logical product term, and the third gate delay is caused by the ORing of the individual terms of the last expression in (10.5).

A commercially available look-ahead carry generator is the SN74S182 device shown in Figure 10.5.

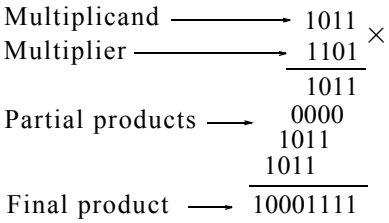
10.7 Binary Multiplication

Binary multiplication with pencil and paper is performed in a similar manner as the decimal multiplication, that is, we obtain the partial products and we shift these partial products before adding to obtain the final product.

Example 10.9

Multiply $(1011)_2$ by $(1101)_2$.

Solution:



The following observations are made from the above example:

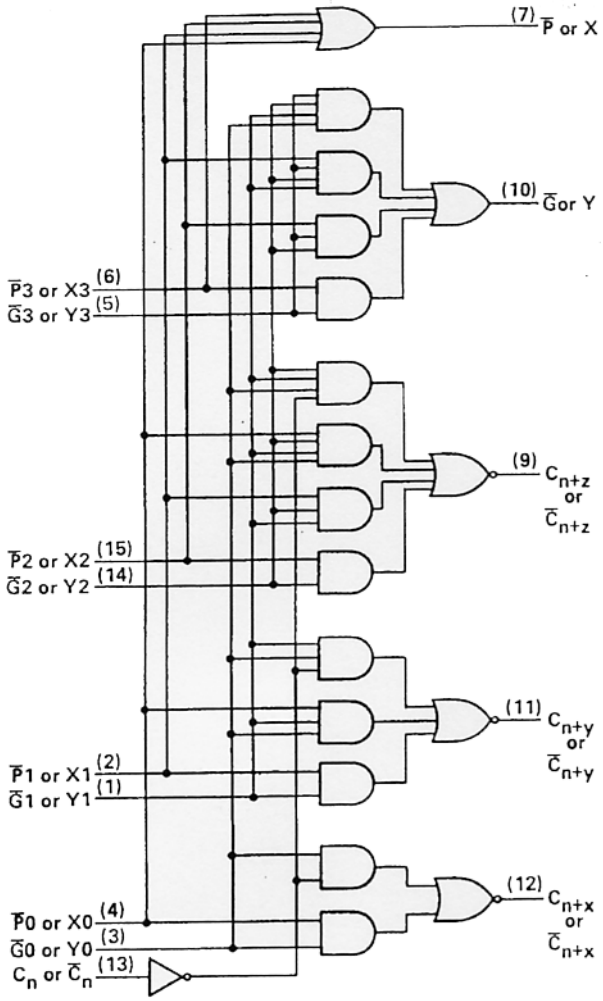

(7) P̄ or X
(10) Ḡ or Y
P̄3 or X3 (6)
Ḡ3 or Y3 (5)
(9) Cn+z or C̄n+z
P̄2 or X2 (15)
Ḡ2 or Y2 (14)
(11) Cn+y or C̄n+y
P̄1 or X1 (2)
Ḡ1 or Y1 (1)
(12) Cn+x or C̄n+x
P̄0 or X0 (4)
Ḡ0 or Y0 (3)
Cn or C̄n (13)

Figure 10.5. Logic diagram of the SN74S182 look-ahead carry generator (Courtesy Texas Instruments)

First, the multiplication starts with the lsb of the multiplier; if it is 1, the first partial product is the multiplicand. If the lsb of the multiplier is 0, the first partial product is zero. This procedure is repeated for the higher bit positions where each subsequent partial product is shifted one bit position to the left from the previous partial product. Then, the final product is obtained by the addition of the shifted partial products.

We recall that multiplication is simply repeated addition. For example, multiplying the number 5 by the number 3 means to add 5 three times. This is the procedure which is employed by the ALU of a typical microprocessor. A typical ALU contains a binary adder which is capable of adding and storing two binary numbers that do not exceed the register capacity. The result (sum) of every addition is stored in the A (Accumulator) register. The number stored in the accumulator is then shifted. Thus, binary multiplication can be performed with binary adders and shift registers as illustrated by the following example.

Example 10.10

Multiply $(1011)_2$ by $(1101)_2$ using addition and shifting operations.

Solution:

$$\begin{array}{ll} \text{Multiplicand} \longrightarrow & 1011 \\ \text{Multiplier} \longrightarrow & \underline{1101} \quad \times \end{array}$$

The contents of the accumulator are first cleared, and since the lsb of the multiplier is 1, the first partial product is added to the contents of the accumulator, which now contains the number 1011 , that is,

$$\text{Accumulator} \longrightarrow 1011$$

Before adding the second partial product, we shift the contents of the accumulator one bit posit ion to the right and thus the accumulator now contains

$$\text{Accumulator} \longrightarrow 01011$$

Since the second multiplier bit is zero, the second partial product is zero, Therefore, the number in the accumulator remains the same and we shift one bit position to the right. The new content of the accumulator is

$$\text{Accumulator} \longrightarrow 001011$$

The third multiplier bit is 1, therefore we add the multiplicand to the accumulator with the alignment shown below.

$$\begin{array}{r} 001011 \\ 1011 \\ \hline 110111 \end{array}$$

The new content of the accumulator is

$$\text{Accumulator} \longrightarrow 110111$$

Before adding the next partial product, we shift contents of accumulator.

$$\text{Accumulator} \longrightarrow 0110111$$

The 4th multiplier bit is 1, therefore we add the multiplicand to we add the multiplicand to the accumulator with the alignment shown below.

$$\begin{array}{r} 0110111 \\ 1011 \\ \hline 10001111 \end{array}$$

There are no more multiplier bits so the product now appears in the accumulator as

Accumulator ⟶ 10001111

IC devices SN74284 and SN74285, when connected together as shown in Figure 10.6, they perform a 4-bit by 4-bit parallel binary multiplication. Their output is an 8-bit product.

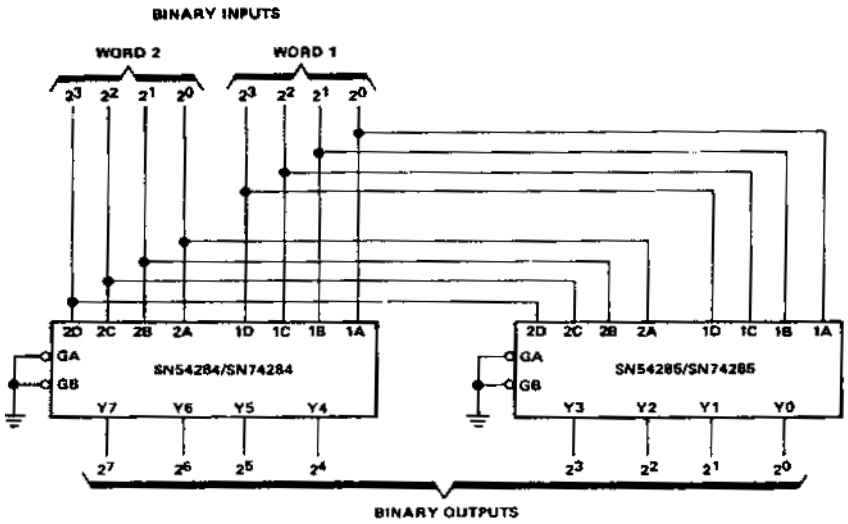


Figure 10.6. 4-bit by 4-bit parallel binary multipliers (Courtesy Texas Instruments)

If the two numbers to be multiplied are positive, they must already be in their true form and since the product is also positive, it is given a sign bit of 0. However, if the numbers to be multiplied are both negative, they will be stored in their 2s complement form, and since the multiplication of two negative numbers produces a positive product which is the same number as if the two numbers were positive, then each of the negative numbers (multiplicand and multiplier) are converted to positive numbers by taking their 2s complement. The product is, of course, kept as a positive number and it is given a sign bit of 0.

When one of the numbers is positive and the other is negative, the product will be negative. Therefore, before multiplication, the negative number is first converted to a positive number by taking its 2s complement. After the multiplication is performed, the product is changed to its 2s complement form with a sign bit of 1.

10.8 Binary Division

The procedure of dividing one binary number (the dividend) by another (the divisor) is simpler than the division of decimal numbers. This is because when we try to determine how many times the divisor goes into the dividend, there are only two possibilities, 0 or 1.

Example 10.11

Divide 1001 by 0011

Solution:

```
                  | 11      Quotient
Divisor 11        | 1001    Dividend
                    11
                  ------
                    0011
                      11
                  ------
                    0000
```

Example 10.12

Divide 1010 by 0100

Solution:

```
                  | 10.1    Quotient
Divisor 100       | 1010    Dividend
                    100
                  -------
                    00100
                      100
                  -------
                    00000
```

A typical ALU carries out the subtractions – which are part of the division procedure – using the 2s complement method, that is, the ALU forms the 2s complement of the subtrahend and performs addition instead of subtraction.

The division of signed numbers is performed in the same manner as multiplication,. Thus negative numbers are first made positive by taking the 2s complement and the division is carried out. If the dividend and the divisor are of the same sign, the quotient is positive and therefore a sign bit of 0 is placed in front of the quotient. If the dividend and the divisor are of opposite sign, the quotient is negative, and therefore the result is changed to a negative number by taking its 2s complement with a sign bit of 1.

Other arithmetic operations such as finding the square, or the square root of a number can be accomplished by certain algorithms, which are series of instructions we must follow to solve a problem. Most microcomputers are provided with software (programs) that include subroutines (programs stored in the memory for use by other programs). These subroutines are used to calculate sines, cosines, tangents, logarithms, squares, square roots, etc.

10.9 Logic Operations of the ALU

Microprocessors include a class of instructions which perform the AND, OR, and XOR logic operations. For instance, the AND operation performs a bit by bit ANDing of the contents of the accumulator and the contents of another register, and stores the result in the accumulator.

Example 10.13

The content of the accumulator is the binary word 00011110 and the content of Register B is the binary word 01001101 . Determine the content of the accumulator after its initial content has been ANDed with the content of Register B.

Solution:

$$\begin{array}{r} 00011110 \\ 01001101 \\ \hline 00001100 \end{array} \quad \text{AND}$$

10.10 Other ALU functions

A typical ALU performs many other functions such as BCD-to-binary, binary-to-BCD, binary-to-Gray, and Gray-to-binary conversions. These conversions can be accomplished by the use of shift registers. Other ALU functions include parity bit generation and parity bit checking. A parity generator is a logic circuit that generates a parity bit, and a parity checker is a logic circuit that checks the parity and produces a logical 1 output whenever an error is detected. In Exercises 13 and 14 at the end of this chapter, it is shown that a parity generator circuit can be implemented with the same circuit as that of a parity checker circuit. A commercially available 9-bit odd/even parity generator/checker is the device SN74180 shown in Figure 10.7.

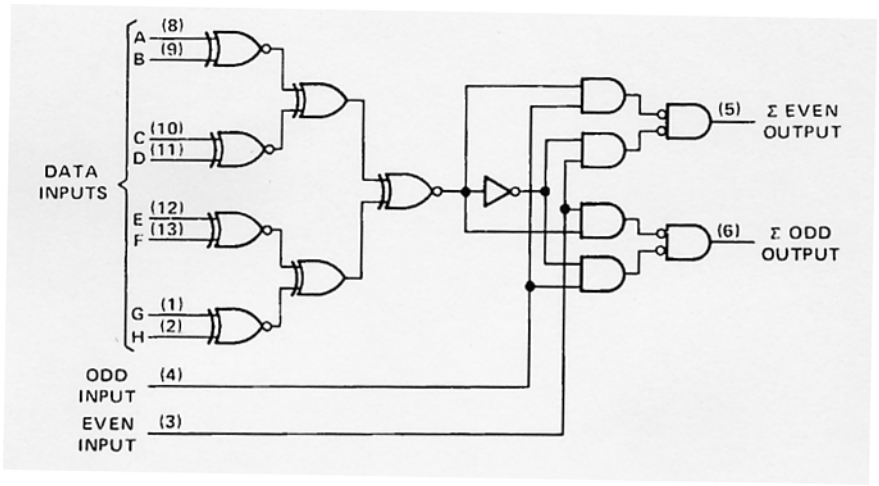


Figure 10.7. 9-bit odd/even parity generator/checker (Courtesy Texas Instruments)

10.11 Summary

- Analog computers are automatic computing devices that operate with electronic signals exhibiting continuous variations of physical quantities.
- Digital computers are automatic computing devices that operate with electronic signals exhibiting discrete variations.
- Computer interfacing is the interconnection and synchronization of digital information transmission between the main computer and the peripherals.
- Computer hardware are the electronic and mechanical parts and materials from which the computer is fabricated.
- Computer software are programs used by a computer.
- Subroutines are programs stored in memory for use by other programs.
- Operating system is a set of programs used to control the running of the user or applications programs.
- A typical digital computer contains an arithmetic/logic unit (ALU), memory unit, control unit, input unit, and output unit. The control unit and the ALU are usually combined into one unit called central processing unit (CPU). When a CPU is contained in a single large scale integration (LSI) chip, it is called a microprocessor.
- A microcomputer is a small digital computer consisting of a microprocessor and other integrated circuits (ICs) comprising the memory, input, and output units.
- Applications are software that are programmed to perform specific tasks. Typical applications are word processors, spreadsheets, databases, internet browsers, and e-mail.
- The BIOS, acronym for basic input/output system, is a set of essential software routines that test hardware at startup, start the operating system, and support the transfer of data among hardware devices. The BIOS is stored in read-only memory (ROM) so that it can be executed when the computer is turned on.
- An integrated circuit, commonly referred to as chip, is a small, thin piece of silicon onto which the transistors making up the microprocessor have been etched.
- A parallel adder is a digital circuit used to add two binary words where each word consists of several bits. Normally, the augend is stored in the accumulator register. A major disadvantage of the parallel adder is the propagation delay of the carry bit from one full adder to the next higher position full adder.
- A serial adder employs only one full adder and the binary words are added one bit at a time. The serial adder has fewer components but it is much slower than a parallel adder.

- Either a serial or a parallel adder can also perform the operation of subtraction. This is accomplished by taking the 2s complement of the subtrahend and thus converting the subtraction operation into an addition operation.
- A practical adder/subtractor circuit can add or subtract negative as well as positive numbers.
- A overflow is a condition in which a calculation produces a unit of data too large to be stored in the location allotted to it. An overflow cannot occur when two numbers of opposite sign are added. An overflow may occur in an addition if the augend and the addend are both positive or both negative.
- The propagation delay problem in a parallel adder circuit is alleviated by a scheme known as look-ahead carry which adds directly the augend A_i with the addend B_i of the *ith* position full adder in the parallel adder circuit and the lsb position carry C_0.
- Binary multiplication and binary division is accomplished by repeated additions and repeated subtractions respectively, and shifting the binary numbers with the use of shift registers.
- Microprocessors can perform a variety of tasks such a logic operations, conversions, parity generation, and parity checking.

10.12 Exercises

1. Design a 4-bit parallel adder/subtractor circuit that uses the 2s complement for subtraction.

For Exercises 2 through 9, all positive numbers are in true form, the negative numbers are in 2s complement form, and the sums or differences are stored in 8-bit registers.

2. Add the numbers +9 and +17 in binary form.

3. Add the numbers –9 and –17 in binary form.

4. Add the numbers –9 and +17 in binary form.

5. Add the numbers +9 and –17 in binary form.

6. Subtract +17 from +9 in binary form.

7. Subtract +9 from +17 in binary form.

8. Subtract –17 from +9 in binary form.

9. Subtract +9 from –17 in binary form.

10. Derive the equations for a 4-bit look-ahead-carry adder circuit.

11. A BCD adder is a combinational logic circuit that adds two BCD numbers and produces a sum which is also in BCD. However, since an invalid BCD representation is obtained whenever the sum is greater than 1 0 0 1, some additional circuitry is required to make the necessary corrections.

a. Construct a table that shows the corresponding binary and BCD sums.

b. Using the table, derive an expression which provides the necessary corrections.

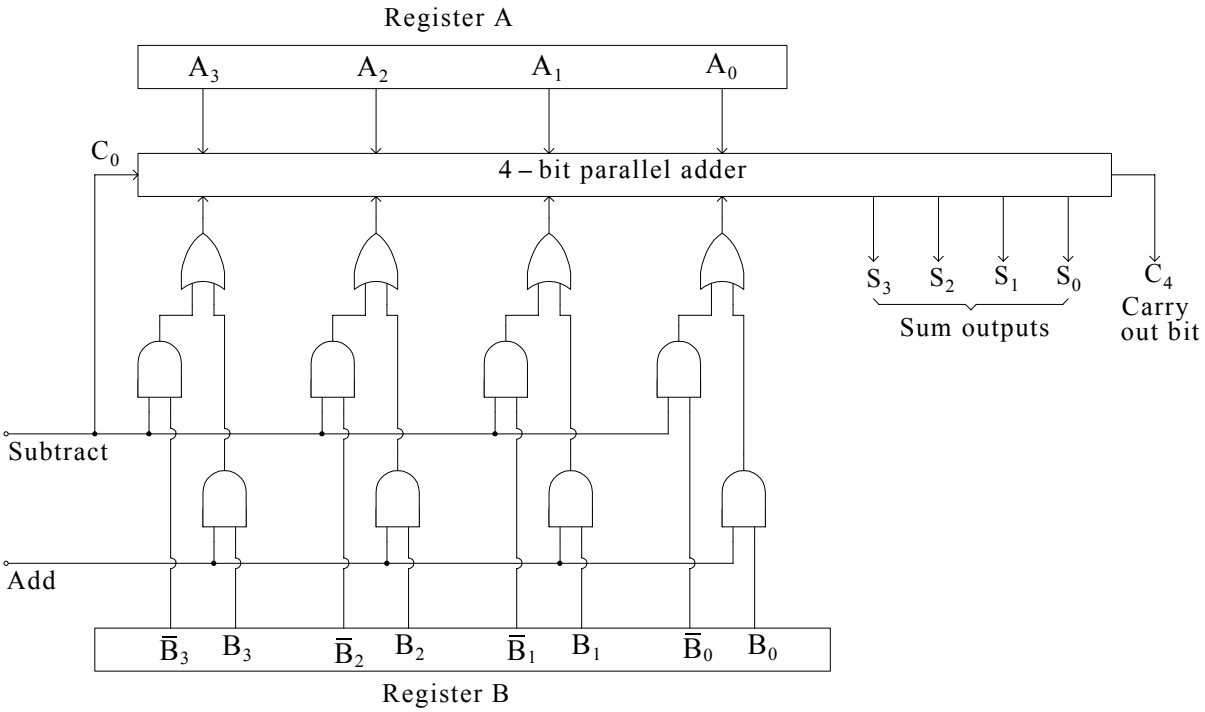
12. The content of the accumulator is the binary word 01011011 and the content of Register B is the binary word 00111100 . Determine the content of the accumulator after its initial content has been XORed with the content of Register B.

13. Design an odd parity bit generator to be used with a 3-bit binary word.

14. Design a parity checker which will check for odd parity in a 4-bit binary word (including the parity bit) and will produce a logical 1 output whenever an error occurs, i.e., when the number of ls in the 4-bit word is even.

10.13 Solutions to End-of-Chapter Exercises

1.



We observe that the adder/subtractor circuit above is provided with both an *add* and *subtract* commands. Therefore, when the add command is High (logic 1), the 4-bit parallel adder adds the Register A contents with the uncomplemented contents of register B. When the subtract command is high, the adder adds the contents of Register A with the complemented contents of register B. Also, whenever the subtract line is High (logic 1), the carry C_0 is also a logic 1 and it is added to the lsb of the sum to form the 2s complement from the 1s complement. The sum outputs are S_0, S_1, S_2, and S_3. The carry output C_4 can be used either as a carry input to another higher position adder/subtractor circuit, or as an overflow bit to indicate that the sum exceeds 1111. This is because the largest signed positive number that can be stored in a 4-bit register is 0111 (+7) and the largest negative number is 1000 (–8). Thus, if the result of an arithmetic operation exceeds these numbers, an overflow condition will result.

2.

$$\begin{array}{rl} (+9) \rightarrow & 00001001 \\ (+17) \rightarrow & \underline{00010001} \quad + \\ (+26) \rightarrow & 00011010 \end{array}$$

The sum is positive and it stored in its true form.

3.

$$\begin{array}{rr} (-9) \rightarrow & 11110111 \\ (-17) \rightarrow & 11101111 \\ \hline (-26) \rightarrow & 111100110 \end{array} +$$

The sum is negative and it stored in its 2s complement form. The 8th position bit is lost.

4.

$$\begin{array}{rr} (-9) \rightarrow & 11110111 \\ (+17) \rightarrow & 00010001 \\ \hline (+8) \rightarrow & 100001000 \end{array} +$$

The sum is positive and it stored in its true form. The 8th position bit is lost.

5.

$$\begin{array}{rr} (+9) \rightarrow & 00001001 \\ (-17) \rightarrow & 11101111 \\ \hline (-8) \rightarrow & 11111000 \end{array} +$$

The sum is negative and it stored in its 2s complement form.

6.

$$\begin{array}{lr} (+9) \text{ leave in true form} \rightarrow & 00001001 \\ (+17) \text{ take 2s complement} \rightarrow & 00010001 \\ \hline (-8) \text{ difference stored in 2s complement} \rightarrow & 11111000 \end{array} +$$

7.

$$\begin{array}{lr} (+17) \text{ leave in true form} \rightarrow & 00010001 \\ (+9) \text{ take 2s complement} \rightarrow & 11110111 \\ \hline (+8) \text{ difference stored in true form} \rightarrow & 100001000 \end{array} +$$

The 8th position bit is lost.

8. Subtract –17 from +9 in binary form.

$$\begin{array}{lr} (+9) \text{ leave in true form} \rightarrow & 00001001 \\ (-17) \text{ take 2s complement of } -17 \rightarrow & 00010001 \\ \hline (+26) \text{ difference stored in true form} \rightarrow & 00011010 \end{array} +$$

9.

$$\begin{array}{lll} (-17) \text{ leave in 2s complement } \rightarrow & 11101111 & + \\ (+9) \text{ take 2s complement of } +9 \rightarrow & \underline{11110111} & \\ (-26) \text{ difference stored in 2s complement} \rightarrow & 111100110 & \end{array}$$

The 8th position bit is lost.

10.

$$\begin{aligned} G_0 &= A_0B_0 \qquad P_0 = A_0 + B_0 \\ G_1 &= A_1B_1 \qquad P_1 = A_1 + B_1 \\ G_2 &= A_2B_2 \qquad P_2 = A_2 + B_2 \\ G_3 &= A_3B_3 \qquad P_3 = A_3 + B_3 \end{aligned}$$

A 4-bit parallel adder with look-ahead-carry may now be implemented from the following relations:

$$\begin{aligned} C_1 &= G_0 + P_0C_0 \\ C_2 &= G_1 + P_1G_0 + P_1P_0C_0 \\ C_3 &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \\ C_4 &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \end{aligned}$$

11.

a. The binary and BCD sums are shown in Table 10.1. The table extends to twenty rows (0 through 19) since the maximum possible sum of two BCD numbers with a possible previous carry is 9 + 9 + I = 19. The binary sums are those which result in a 4-bit binary adder using the rules of binary addition. The weights of binary addition are represented as X_8, X_4, X_2, and X_1, and the carry is denoted as Y to distinguish them from the weights S_8, S_4, S_2, and S_1, and the carry C of the corrected BCD summation.

b. The table shows that the binary and BCD sums are identical whenever the sum is equal to or less than 1 0 0 1 and therefore no correction is required. However, when the binary sum is 1 0 1 0 or greater we must add 0 1 1 0 to obtain the correct BCD representation. The expression that provides the necessary corrections is derived directly from the table by observing that a correction is required when:

$$\begin{aligned} Y &= 1 \\ X_8X_4 &= 1 \\ X_8X_2 &= 1 \end{aligned}$$

Therefore, the condition for correction C, is

$$C = Y + X_8X_4 + X_8X_2$$

TABLE 10.1 BCD addition

Decimal	Binary Sum					BCD Sum				
	Y	X_8	X_4	X_2	X_1	C	S_8	S_4	S_2	S_1
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1
2	0	0	0	1	0	0	0	0	1	0
3	0	0	0	1	1	0	0	0	1	1
4	0	0	1	0	0	0	0	1	0	0
5	0	0	1	0	1	0	0	1	0	1
6	0	0	1	1	0	0	0	1	1	0
7	0	0	1	1	1	0	0	1	1	1
8	0	1	0	0	0	0	1	0	0	0
9	0	1	0	0	1	0	1	0	0	1
10	0	1	0	1	0	1	0	0	0	0
11	0	1	0	1	1	1	0	0	0	1
12	0	1	1	0	0	1	0	0	1	0
13	0	1	1	0	1	1	0	0	1	1
14	0	1	1	1	0	1	0	1	0	0
15	0	1	1	1	1	1	0	1	0	1
16	1	0	0	0	0	1	0	1	1	0
17	1	0	0	0	1	1	0	1	1	1
18	1	0	0	1	0	1	1	0	0	0
19	1	0	0	1	1	1	1	0	0	1

A commercially available 4-bit BCD adder is the Philips Semiconductor 74F583 device shown below.

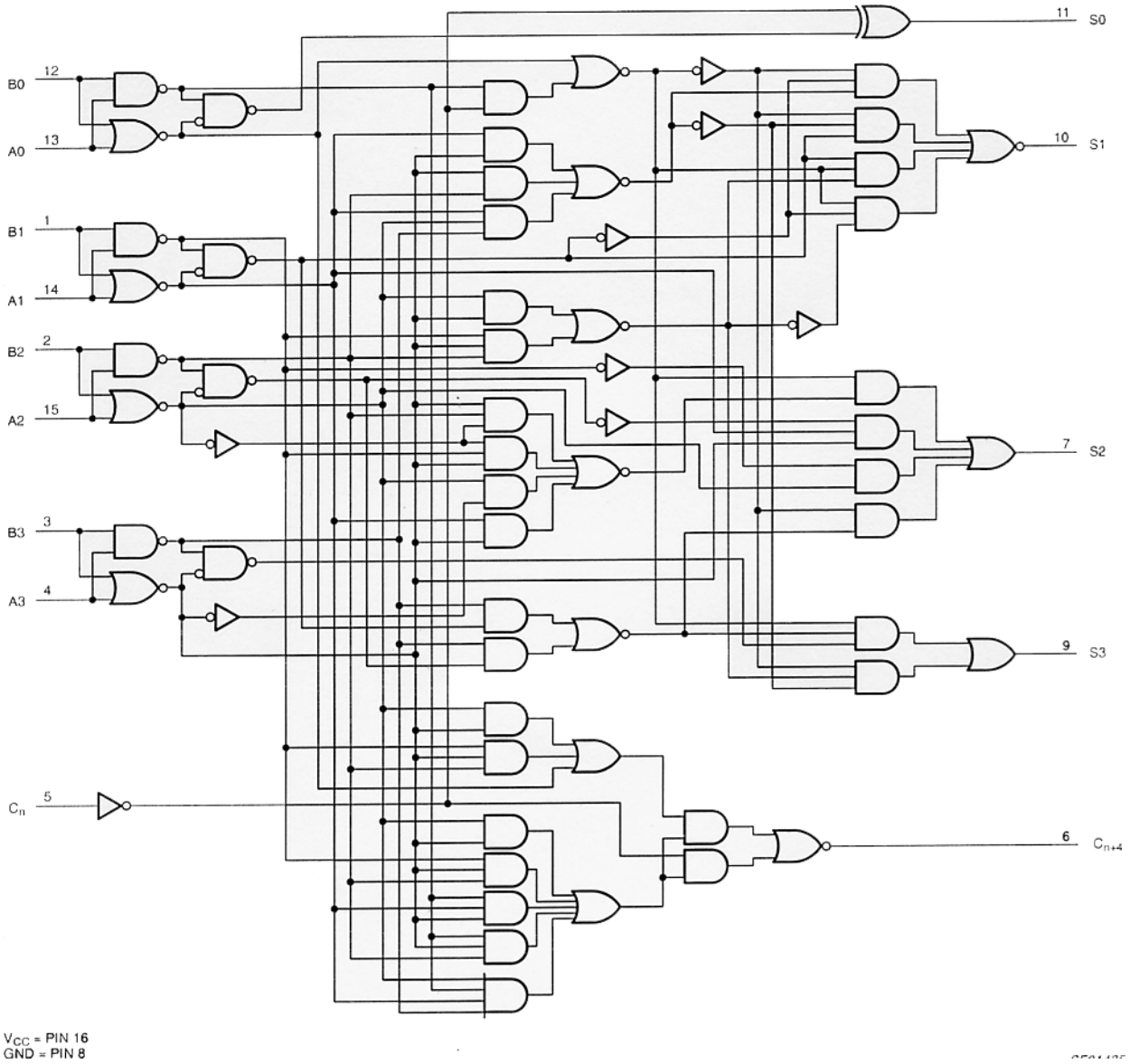


Figure 10.8. 4-bit BCD adder (Courtesy Philips Semiconductor)

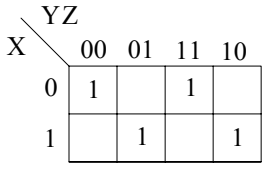
12.

$$\begin{array}{r} 01011011 \\ 00111100 \\ \hline 01100111 \end{array} \text{ XOR}$$

13.

Let the 3-bit binary word be represented by the variables X, Y, and Z, and the generated parity bit by P. We construct the following truth table and K-map.

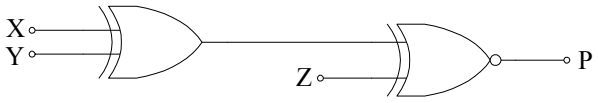
Inputs			Output
X	Y	Z	P
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0



The K-map shows that no simplification is possible; therefore,

$$P = \overline{X}\overline{Y}\overline{Z} + \overline{X}YZ + X\overline{Y}Z + XY\overline{Z} = (XY + \overline{X}\overline{Y})\overline{Z} + (\overline{X}Y + X\overline{Y})Z$$
$$= \overline{(X \oplus Y)}\overline{Z} + (X \oplus Y)Z$$

The circuit implementation is shown below.



14.

Let the 4-bit binary word be represented by the variables X, Y, and Z, and the parity bit by P. Let the output be denoted as C. We construct the following truth table and K-map.

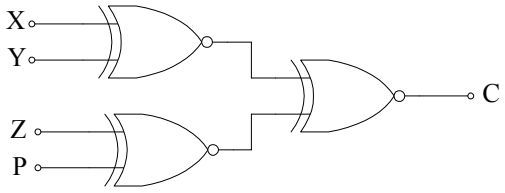
Inputs				Output
X	Y	Z	P	C
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
0	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

XY \ ZP	00	01	11	10
00	1		1	
01		1		1
11	1		1	
10		1		1

The K-map shows that no simplification is possible; therefore,

$$\begin{aligned} P &= \overline{X}\overline{Y}\overline{Z}\overline{P} + \overline{X}\overline{Y}ZP + \overline{X}Y\overline{Z}P + \overline{X}YZ\overline{P} + XY\overline{Z}\overline{P} + XYZP + X\overline{Y}\overline{Z}P + X\overline{Y}Z\overline{P} \\ &= \overline{X}\overline{Y}(\overline{Z}\overline{P} + ZP) + \overline{X}Y(\overline{Z}P + Z\overline{P}) + XY(\overline{Z}\overline{P} + ZP) + X\overline{Y}(\overline{Z}P + Z\overline{P}) \\ &= (\overline{X}\overline{Y} + XY)(\overline{Z}\overline{P} + ZP) + (\overline{X}Y + X\overline{Y})(\overline{Z}P + Z\overline{P}) \end{aligned}$$

The circuit implementation is shown below.



By comparison of the parity generator circuit of Exercise 13 and the parity checker circuit above, we see that the parity generator circuit can be implemented with the parity checker circuit if the input P is kept at logic 0 and the output is denoted as P.

Chapter 11

Introduction to Field Programmable Devices

This chapter is an introduction to Field Programmable Devices (FPDs) also referred to as Programmable Logic Devices (PLDs). It begins with the description and applications of Programmable Logic Arrays (PLAs), continues with the description of Simple PLDs (SPLDs) and Complex PLDs (CPLDs), and concludes with the description of Field Programmable Gate Arrays (FPGAs).

11.1 Programmable Logic Arrays (PLAs)

A *Programmable Logic Array* (*PLA*) is a small *Field Programmable Device* (*FPD*) that contains two levels of logic, AND and OR, commonly referred to as AND-plane and OR-plane respectively. In concept, a PLA is similar to a ROM but does not provide full decoding of the variables, in other words, a PLA does not generate all the minterms as a ROM does.* Of course, a ROM can be designed as a combinational circuit with any unused words as don't care conditions, but the unused words would result in a poor design. Consider, for example, the conversion of a 16-bit code into an 8-bit code. In this case, we would have $2^{16} = 65536$ input combinations and only $2^8 = 256$ output combinations, and since we would only need 256 valid entries for the 16-bit code, we would have $65536 - 256 = 65280$ words wasted. With a PLA we can avoid this waste.

The size of a PLA is defined by the number of inputs, the number of product terms in the AND-plane, and the number of the sum terms in the OR-plane. Figure 11.1 shows the block diagram of a typical PLA. As shown, it consists of n inputs, m outputs, k product terms, and m sum terms. The group of the k terms constitutes the AND-gate plan, and the group of m terms constitutes the OR-gate plane. Links, shown as fuses, are inserted at all inputs and their complemented values to each of the AND gates. Links are also provided between the outputs of the AND gates and the inputs of the OR gates. Another set of links appears at the output lines that allows the output function to appear either in the AND-OR form or the AND-OR-Invert form. These forms are shown in Figure 11.2. The inverters at the output section of the PLA are tri-state devices.†

---

* *We recall that a ROM contains 2^n words and m bits per word where n is the number of inputs in a decoder circuit.*

† *Tri-state devices have three outputs, logic 0, logic 1, and Hi-Z. For a detailed discussion on tri-state devices, please refer to electronic devices and amplifier circuits, ISBN 0-9744239-4-7.*

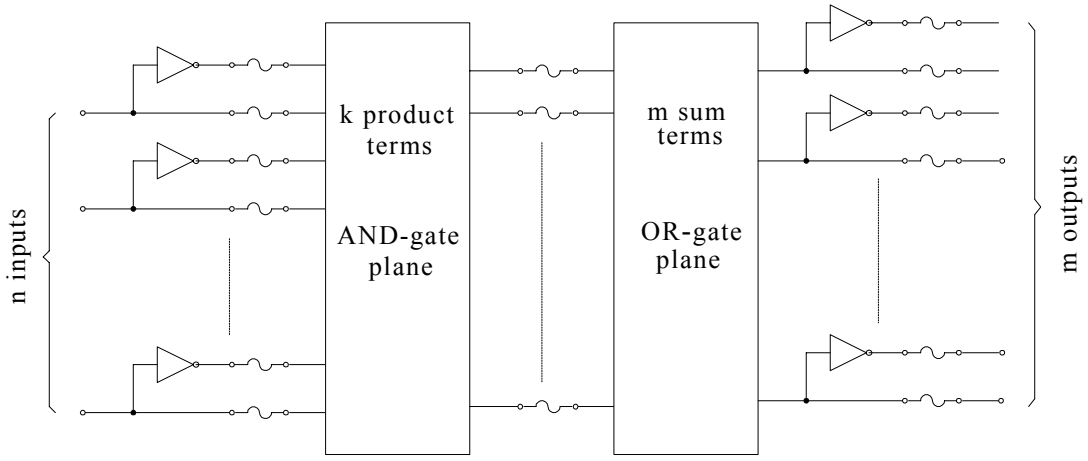


Figure 11.1. Block diagram of a typical PLA

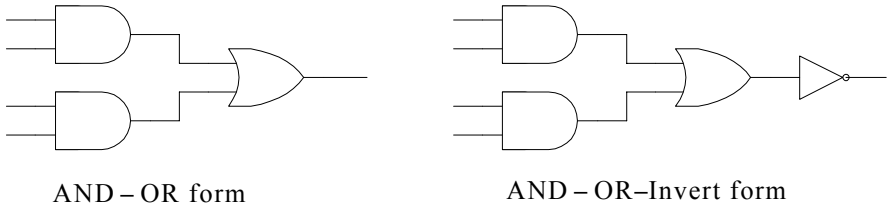


Figure 11.2. AND-OR and AND-OR-Invert forms

PLAs may be *mask-programmable* or *field programmable*. With a mask programmable PLA the buyer submits a PLA program table to the manufacturer. With a field programmable PLA, the buyer configures the PLA himself. The following examples illustrate how a PLA is programmed.

Example 11.1

A combinational circuit is defined as the functions

$$F_1 = A\bar{B}\bar{C} + A\bar{B}C + ABC$$

$$F_2 = \bar{A}BC + A\bar{B}C + ABC$$

Implement the digital circuit with a PLA having 3 inputs, 3 product terms, and 2 outputs.

Solution:

The K-maps for F_1 and F_2 are shown in Figure 11.3. These maps yield the product terms $A\bar{B}$, AC, and BC. Thus, the circuit can be implemented with a PLA that has 3 inputs, A, B, and C, and two outputs F_1 and F_2 as shown in Figure 11.4.

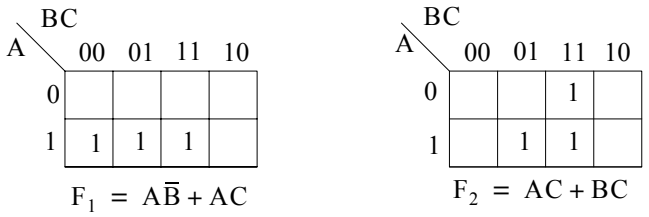


Figure 11.3. K-maps for the functions of Example 11.1

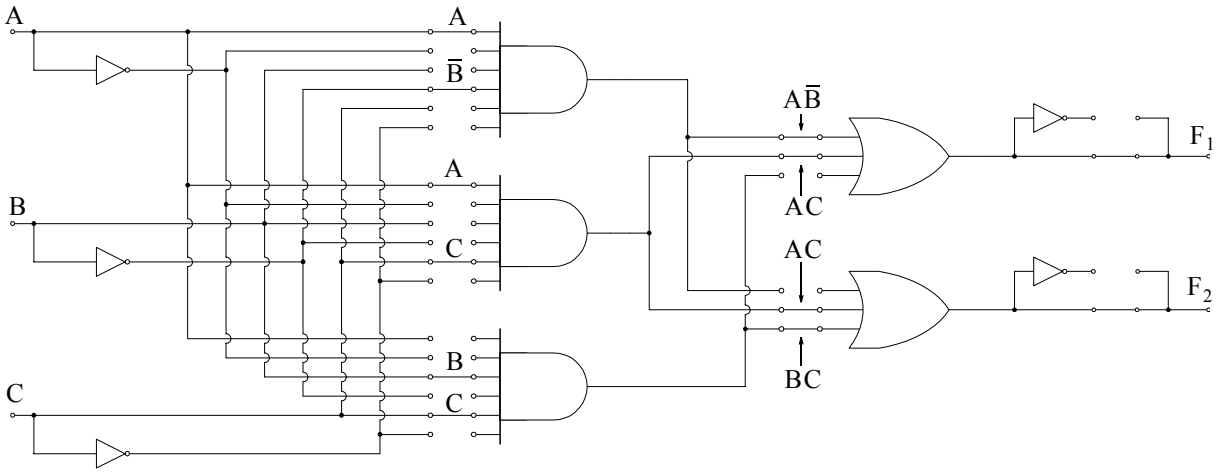


Figure 11.4. PLA for Example 11.1

Programming the PLA implies specifying the paths in its AND-OR-Invert form. The PLA program for this example is as shown in Table 11.1

TABLE 11.1 PLA Program table for Example 11.1

	Product Term	Inputs			Outputs		
		A	B	C	F_1	F_2	
$A\overline{B}$	1	1	0	—	1	—	
AC	2	1	—	1	1	1	
BC	3	—	1	1	—	1	
					T	T	T/C

As shown in Table 11.1, the paths between the inputs and the AND gates are specified under the Inputs column where a logic 1 specifies a path from the corresponding input to the input of the AND gate that forms the product term. A logic 0 in the input column specifies a path from the corresponding complemented input to the input of the AND gate. A long dash (—) indicates no connection. Thus, the unwanted links are broken and the remaining form the desired paths as shown in Figure 11.4. We have assumed that the open links at the inputs of the AND gates behave as logic 1.

The paths between the AND and or gates are specified under the Outputs column. The output variables are labeled with a logic 1 for all of the product terms that form the function. For instance, the first function is $F_1 = A\overline{B} + AC$, and thus F_1 is marked with logic 1s for the product terms 1 $(A\overline{B})$and 2 (AC), and with a dash (—) for the product term 3 (BC). Each product term that has a logic 1 under the Output column requires a path from the corresponding AND gate to the output of the OR gate, and those marked with a dash (—) indicate no path.

In Table 11.1, the T/C notation stands for True/Complement and since both functions F_1 and F_2 appear in the uncomplemented (true) form, both functions are marked with the T notation. We have assumed that the open links at the inputs of the OR gates behave as logic 0.

When designing digital systems with PLAs, there is no need to show the internal connections as we did in Figure 11.4; all is needed is a PLA program table such as that of Table 11.1. Also, an attempt should be made to reduce the total number of distinct product terms so that the PLA would have a number of AND terms in their simplest form. It is a good design practice to examine both the true and complemented expressions to find out which provides product terms that are common to the other functions. Example 11.2 below illustrates this point.

Example 11.2

A combinational circuit is defined as the functions

$$F_1 = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

$$F_2 = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + ABC$$

Implement the circuit with a PLA having 3 inputs, 4 product terms, and 2 outputs.

Solution:

We begin by forming K-maps for both the true and complemented forms of the given functions, that is, F_1, $\overline{F}_1$, F_2, and $\overline{F}_2$ as shown in Figure 11.5. We observe that the selection of $\overline{F}_1$ and F_2 yields only 4 distinct product terms, that is, $\overline{A}\,\overline{B} + \overline{A}\,\overline{C} + \overline{B}\,\overline{C} + ABC$. The PLA program table for this combination of the input variables is shown in Table 11.2. The table is consistent with the relations

$$F_1 = \overline{\overline{A}\,\overline{B} + \overline{A}\,\overline{C} + \overline{B}\,\overline{C}}$$

$$F_2 = \overline{A}\,\overline{C} + \overline{B}\,\overline{C} + ABC$$

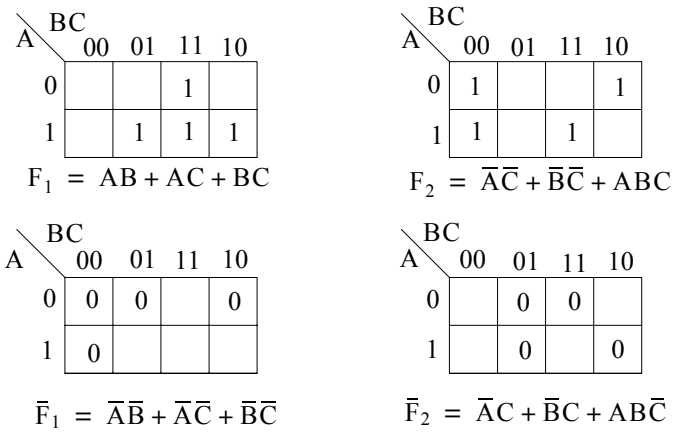


Figure 11.5. K-maps for the functions of Example 11.2

	Product Term	Inputs			Outputs	
		A	B	C	F_1	F_2
$\overline{A}\,\overline{B}$	1	0	0	—	1	1
$\overline{A}\,\overline{C}$	2	0	—	0	1	1
$\overline{B}\,\overline{C}$	3	—	0	0	1	—
ABC	4	1	1	1	—	1
					C	T	T/C

The circuits for Examples 11.1 and 11.2 are very small and implementation with PLAs is impractical. It would be more practical to use small scale integration combinational circuits such as those in Chapter 7. The first PLAs, introduced in the 1970s, contained 48 product terms (AND terms) and 8 sum terms (OR terms). The manufacturing costs of these PLAs were high, and because of the two levels of logic (AND and OR), the speed was poor. Typical applications included code conversion, peripheral controllers, function generators, address mapping, fault detectors, and frequency synthesizers.

11.2 Programmable Array Logic (PAL)

The *Programmable Array Logic* (*PAL*) devices were developed to overcome the deficiencies of PLAs. A major difference between PLAs and PALs is that the latter is designed with a programmable wired-AND* plane followed by a fixed number of OR gates and flip flops so that sequential circuits could also be implemented. Variants of the original PALs were designed with different

* *For a description of wired-AND devices, please refer to Electronic Devices and Amplifier Circuits, ISBN 0-9744239-4-7.*

inputs, and various sizes of OR-gates, and they form the basis for present-day digital hardware designs. Figure 11.6 shows the block diagram of a typical PAL.

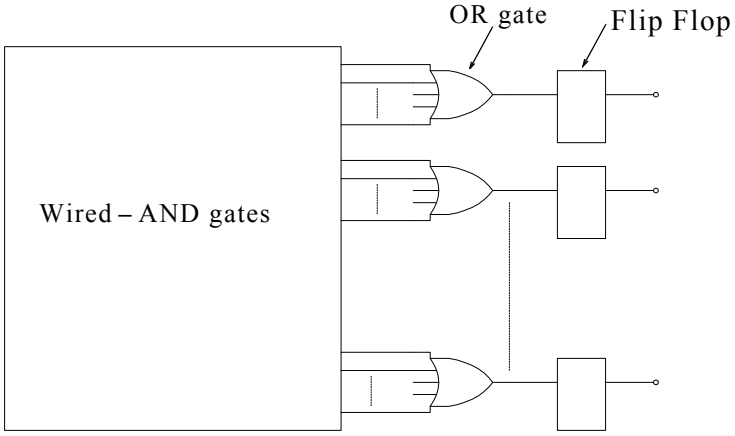


Figure 11.6. Block diagram of a typical PAL

PLAs, PALs, and PAL variants are generally referred to as *Simple Programmable Logic Devices* (*SPLDs*). Four popular SPLDs are the 16R8, 16R6, 16R4, and 16L8 series. These devices have potentially 16 inputs and 8 outputs configurable by the user. Output configurations of 8 registers (flip flops), 8 combinational, 6 registers and 2 combinational, 4 registers and 4 combinational, or 8 combinational are provided where the letter R refers to the number of "registered" outputs and the letter L to non-registered, that is, combinational outputs. Figure 11.7 shows the Cypress Semiconductor functional variations of this series.

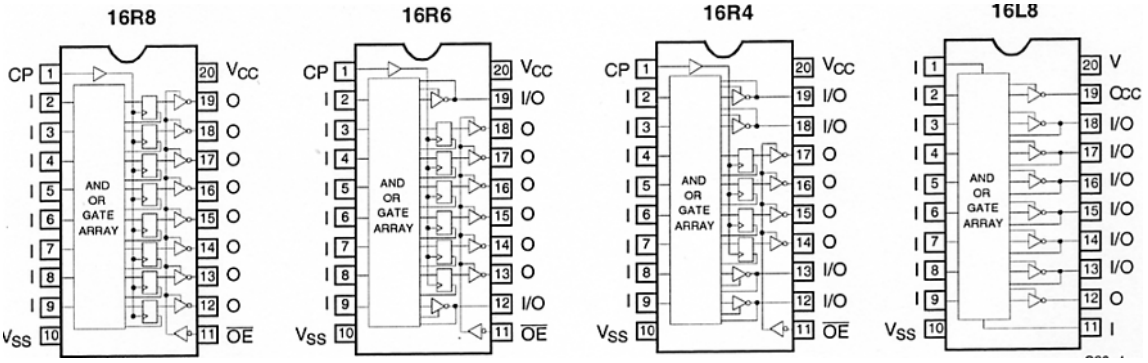


Figure 11.7. The 16R and 16L series SPLDs (Courtesy Cypress Semiconductor)

11.3 Complex Programmable Logic Devices (CPLDs)

Complex Programmable Logic Devices (CPLDs) integrate multiple SPLDs into a single chip. A typical CPLD provides a logic capacity equivalent to about 50 single SPLDs. Altera, the world's second-largest manufacturer – presently, the largest manufacturer is Xilinx – of programmable semiconductors, introduced the MAX 5000, MAX 7000, and MAX 9000 series of CPLDs. MAX 5000 is based on an older technology, the MAX 7000 family is a high-performance PLD and it is

fabricated with advanced CMOS technology designed with state-of-the art logic capacity and speed performance, and MAX 9000 is very similar to MAX 7000 except that the it has higher logic capacity.

11.3.1 The Altera MAX 7000 Family of CPLDs

Because of its versatility and popularity, our discussion on Altera's CPLD will be based on the MAX 7000 family of CPLDs.

The MAX 7000 family consists of seven EEPROM-based PLDs providing 600, 1250, 1800, 2500, 3200, 3750, and 5000 usable gates, *In System Programmability* (*ISP*)* circuitry compatible with IEEE Std 1532†, and only 5 ns pin-to-pin logic delay. MAX 7000 devices use CMOS EEPROM cells to implement logic functions and accommodate a variety of independent combinational and sequential functions. The devices can be reprogrammed for quick and efficient iterations during design development and debug cycles, and can be programmed and erased up to about 100 times.

Because of their complexities, when designing digital circuits for implementation with FPDs, it is necessary to employ Computer-Aided Design (CAD) programs. For the MAX 7000 family of PLDs, software design support is provided by Altera's development system for Windows-based platforms, Sun's SPARC Station, and HP 9000 Series 700/800 workstations.

The MAX 7000 architecture consists of the following elements:

1. Logic Array Blocks (LABs)
2. Macrocells
3. Expander product terms
4. Programmable Interconnect Array (PIA)
5. I/O control blocks.

Figure 11.8 shows the basic architecture of Altera's MAX 7000 family of CPLDs. As shown, it contains an array of blocks referred to as *Logic Array Blocks* (*LABs*), a *Programmable Interconnect Array* (*PIA*) that contains interconnect wires, and *I/O control blocks*. Each LAB contains 16 macrocells and the PIA can connect any LAB input or output to any other LAB. It also includes four dedicated inputs that can be used as general-inputs or a high-speed, global control signals (clock, clear), and two output enable signals.

Figure 11.9 shows a MAX 7000 macrocell. The Logic Array shown on the left of Figure 11.9 provides five product terms per macrocell and the Product-Term Select Matrix allocates these product terms for use as either primary logic inputs (to the OR and XOR gates) to implement combi-

* *The system programmability is available only in the MAX 7000S family of Altera CPLDs. Whereas the MAX 7000 family devices can be programmable in an out-of-circuit special programming unit, the MAX 7000S devices are programmable with a built-in IEEE Std 1149.1 Joint Test Action Group (JTAG) interface.*

† *This is a standard for programmable devices such as PROMs, CPLDs, and FPGAs.*

national functions, or as secondary inputs to the macrocell's register clear, preset, clock, and clock enable control functions. Two kinds of expander product terms, referred to as *expanders*, are available to supplement macrocell logic resources:

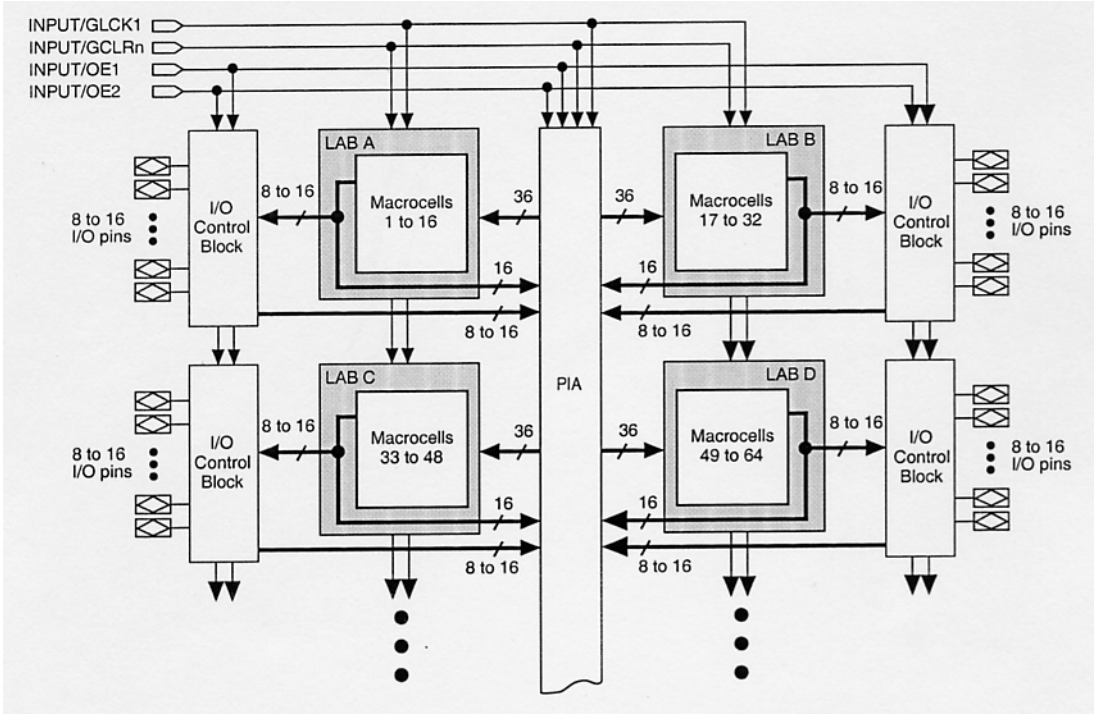


Figure 11.8. Typical MAX 7000 device block diagram (Courtesy Altera Corporation)

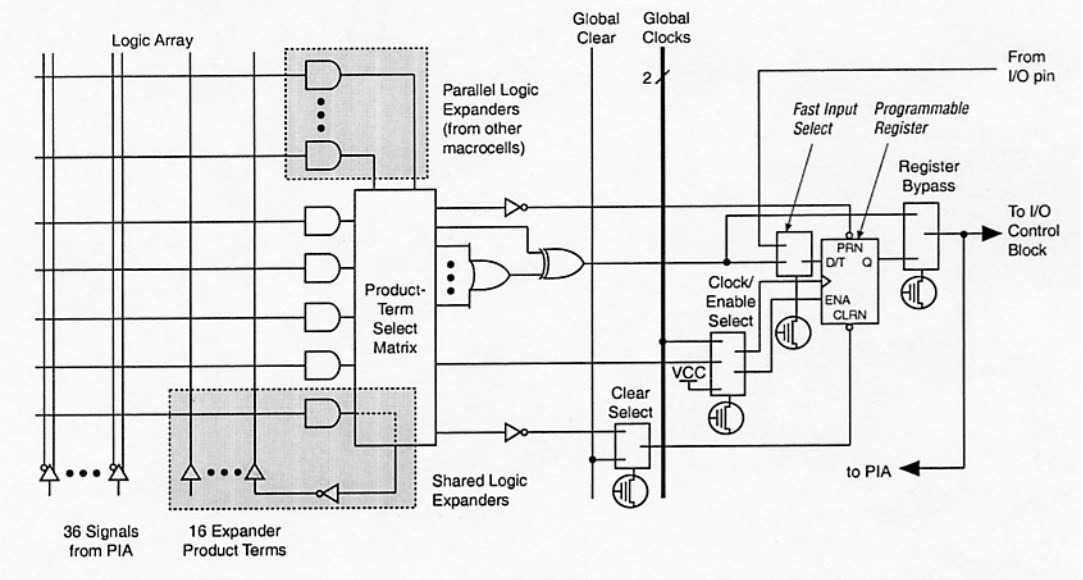


Figure 11.9. MAX 7000 Macrocell (Courtesy Altera Corporation)

1. *Shareable expanders* – inverted product terms that are fed back into the logic array

2. *Parallel expanders* – product terms borrowed from adjacent macrocells

Product-term allocation is automatically optimized according to the logic requirements of the design by the Altera development system. For registered functions, each macrocell flip flop can be individually programmed to implement D, T, JK, or SR operation with programmable clock control. The flip flop can be bypassed for combinational operation. During design entry, the designer specifies the desired flip flop type; the Altera development software then selects the most efficient flip flop operation for each registered function to optimize resource utilization. The expanders make the MAX 7000 family of CPLDs more efficient in chip area since typical logic functions normally do not require more than five product terms. But this architecture supports more than five if needed

Shareable Expanders

Each LAB has 16 *shareable expanders* that can be viewed as a pool of uncommitted single product terms (one from each macrocell) with inverted outputs that feed back into the logic array. Each shareable expander can be used and shared by any or all macrocells in the LAB to build complex logic functions. Figure 5 shows how shareable expanders can feed multiple macrocells.

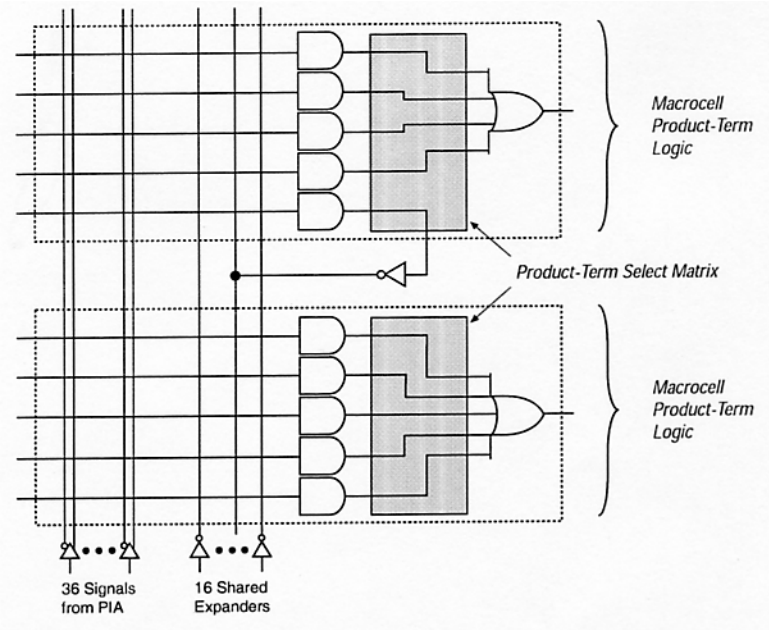


Figure 11.10. Shareable Expanders in the MAX 7000 family of CPLDs (Courtesy Altera Corporation)

Parallel Expanders

Parallel expanders are unused product terms that can be allocated to a neighboring macrocell to implement fast, complex logic functions. Parallel expanders allow up to 20 product terms to directly feed the macrocell OR logic, with five product terms provided by the macrocell and 15 parallel expanders provided by neighboring macrocells in the LAB. The Altera development sys-

tem can allocate up to three sets of up to five parallel expanders automatically to the macrocells that require additional product terms. For instance, if a macrocell requires 14 product terms, the system uses the five dedicated product terms within the macrocell and allocates two sets of parallel expanders; the first set includes five product terms, and the second set includes four product terms.

Two groups of 8 macrocells within each LAB (e.g., macrocells 1 through 8 and 9 through 16) form two chains to lend or borrow parallel expanders. A macrocell borrows parallel expanders from lower-numbered macrocells. For example, macrocell 8 can borrow parallel expanders from macrocell 7, from macrocells 7 and 6, or from macrocells 7, 6, and 5. Within each group of 8, the lowest-numbered macrocell can only lend parallel expanders and the highest-numbered macrocell can only borrow them. Figure 11.11 shows how parallel expanders can be borrowed from a neighboring macrocell.

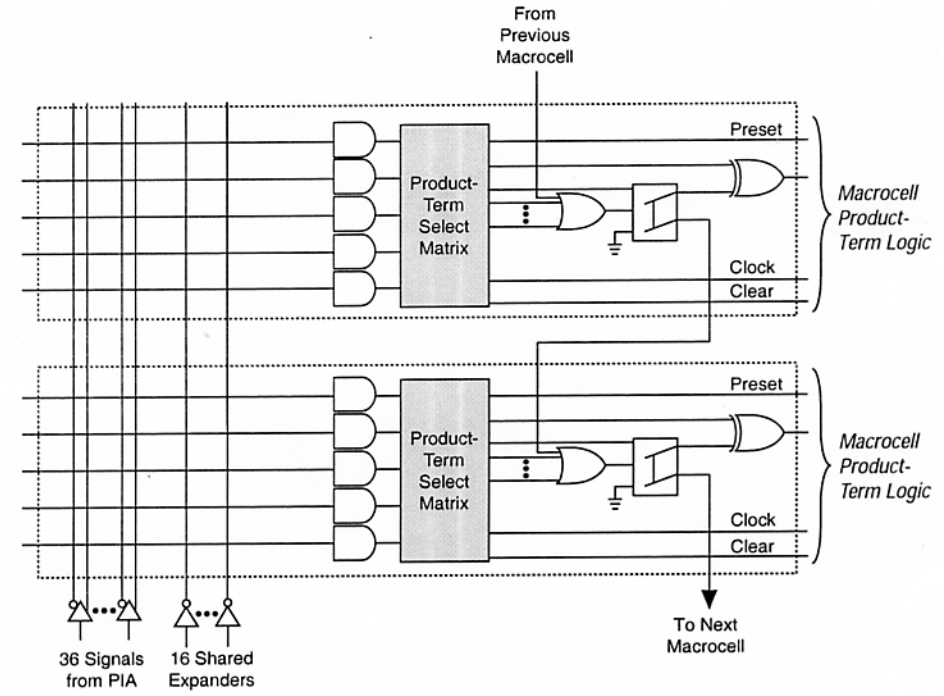


Figure 11.11. Parallel Expanders in the MAX 7000 family of CPLDs (Courtesy Altera Corporation)

Programmable Interconnect Array

The *Programmable Interconnect Array* (*PIA*) is a global bus with programmable paths that connect any signal source to any destination on the device. Logic is routed between LABs via the PIA. All MAX 7000 dedicated inputs, I/O pins, and macrocell outputs feed the PIA which makes the signals available throughout the entire device. Figure 11.12 shows how the PIA signals are routed into the LAB. As shown, an EEPROM cell controls one input to a 2-input AND gate, which selects a PIA signal to drive a LAB. Whereas in other PLDs the routing delays of channel-based

routing schemes are cumulative, variable, and path-dependent, the MAX 7000 PIA has a fixed delay. The PTA thus eliminates skew between signals and makes timing performance easy to predict.

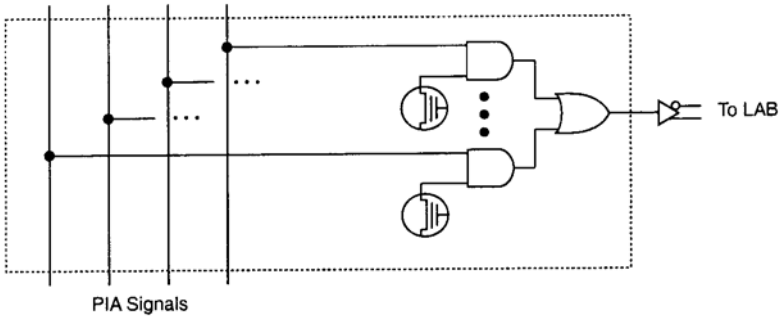


Figure 11.12. PIA routing in the MAX 7000 family of CPLDs (Courtesy Altera Corporation)

I/O Control Blocks

The I/O control block allows each I/O pin to be individually configured for input, output, or bidirectional operation. All I/O pins have a tri-state buffer that is individually controlled by one of the global output enable signals or directly connected to ground to the power supply. Figure 11.13 shows the I/O control block for the MAX 7000 family.

In System Programmability (ISP)

As stated earlier, MAX 7000S devices are in-system programmable via the industry-standard Joint Test Action Group (JTAG) interface (IEEE Std. 1149.1-1990). ISP allows quick, efficient iterations during design development and debugging cycles. The MAX 7000S architecture internally generates the high programming voltage required to program EEPROM cells, allowing in-system programming with only a single 5.0 V power supply.

In-system programming can be accomplished with either an adaptive or constant algorithm. An adaptive algorithm reads information from the unit and adapts subsequent programming steps to achieve the fastest possible programming time for that unit. Because some in-circuit testers cannot support an adaptive algorithm, Altera offers devices tested with a constant algorithm.

During in-system programming, instructions, addresses, and data are shifted into the MAX 7000S device through the TDI input pin. Data is shifted out through the TDO output pin and compared against the expected data.

Programming a pattern into the device requires the following six ISP steps. A stand-alone verification of a programmed pattern involves only steps 1, 2, 5, and 6.

1. Enter ISP – The enter ISP stage ensures that the I/O pins transition smoothly from user mode to ISP mode. This step lasts about 1 ms.
2. Check ID – Before any program or verify process, the silicon ID is checked. The time required to read this silicon ID is relatively small compared to the overall programming time.

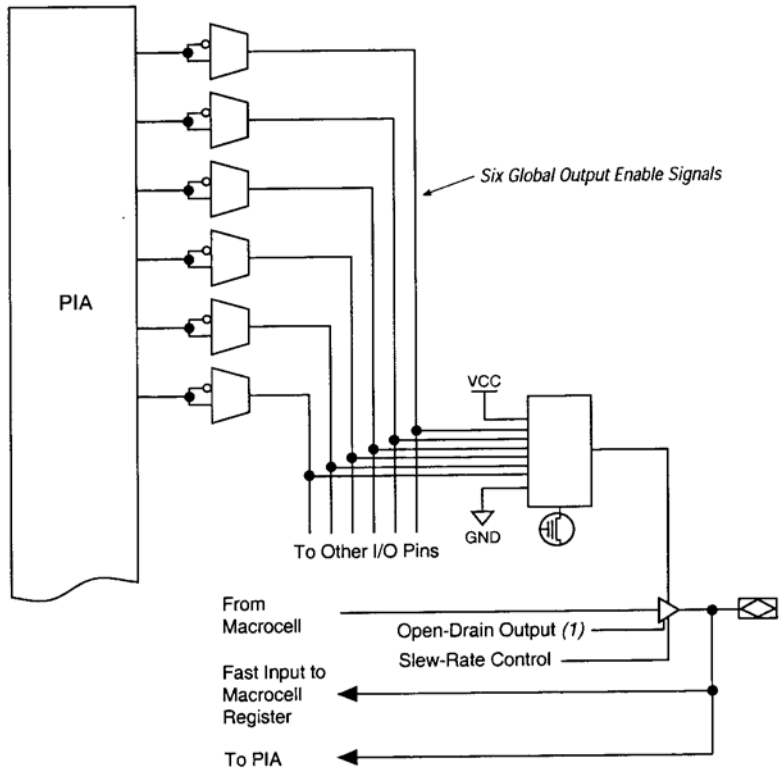


Figure 11.13. PIA routing in the MAX 7000 family of CPLDs (Courtesy Altera Corporation)

3. Bulk Erase – Erasing the device in-system involves shifting in the instructions to erase the device and applying one erase pulse of 100 ms.

4. Program – Programming the device in-system involves shifting in the address and data and then applying the programming pulse to program the EEPROM cells. This process is repeated for each EEPROM address.

5. Verify – Verifying an Altera device in-system involves shifting in addresses, applying the read pulse to verify the EEPROM cells, and shifting out the data for comparison. This process is repeated for each EEPROM address.

6. Exit ISP – An exit ISP stage ensures that the I/O pins transition smoothly from ISP mode to user mode. The exit ISP step requires 1 ms.

11.3.2 The AMD Mach Family of CPLDs

The Advanced Micro Devices (AMD) family of CPLDs consists of the *Mach 1* through *Mach 5* series where each Mach device comprises several PALs. Mach 1 and Mach 2 are made up of 22V16 PALs, Mach 3 and Mach 4 are made up of 34V16 PALs, and Mach 5 is also made of

34V16 PALs but is it faster. All Mach devices are based on EEPROM technology. We will only discuss the Mach 4 since it is the most popular among this series. Its architecture is shown in Figure 11.14.

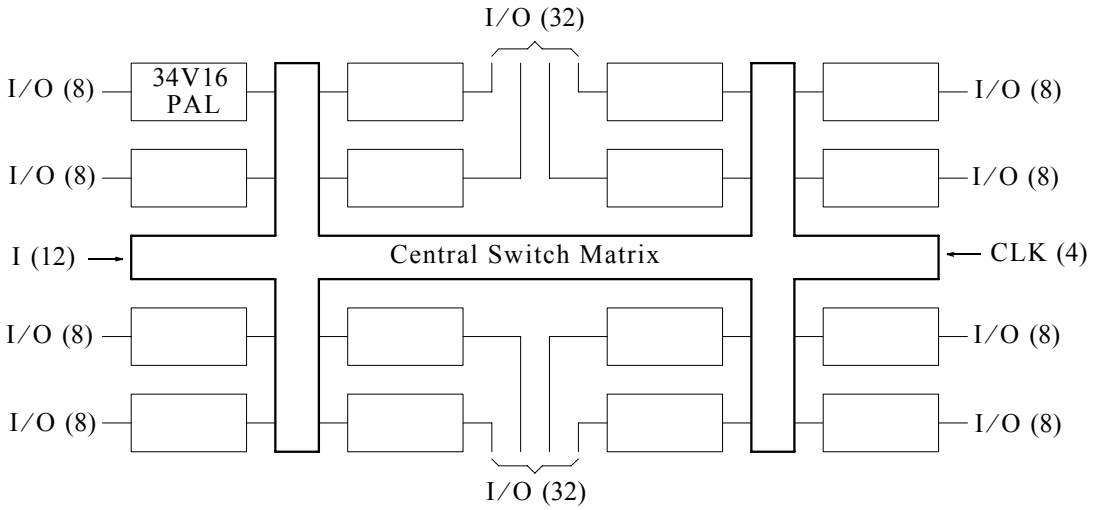


Figure 11.14. Architecture of the AMD Mach 4 CPLD (Courtesy Advanced Micro Devices)

The designation 34V16 indicates that this PAL has 34 inputs and 16 outputs, and the letter V stands for versatile, that is, each output can be configured either as registered (flip flops) or combinational circuitry. The *Central Switch Matrix* is an interconnect bus that allows connecting all 34V16 PALs together and thus the timing delays are predictable. A block diagram of the 34V16 PAL is shown in Figure 11.15.

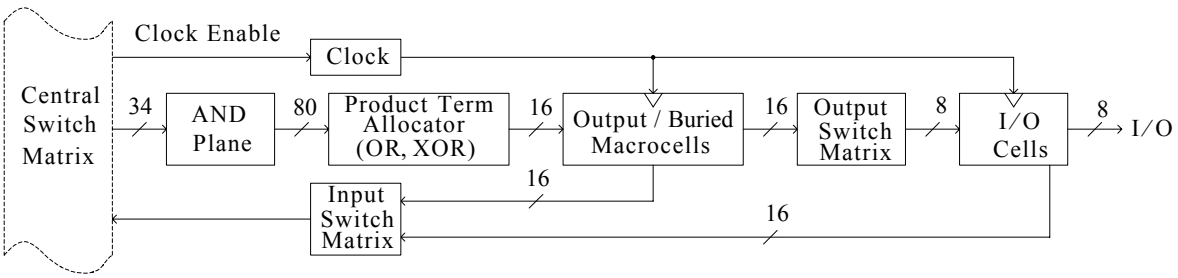


Figure 11.15. Block diagram of the AMD 34V16 PAL (Courtesy Advanced Micro Devices)

In Figure 11.15, the product term allocator distributes and shares product terms from the AND plane to those OR gates that require them. We recall from our previous discussion that a conventional PAL has a fixed-size OR gates plane; therefore, the 34V16 PAL provides more flexibility. The macrocells can either be I/O macrocells, which include an I/O cell which is associated with an I/O pin, or *buried macrocells*, which do not connect to an I/O. The combination of I/O Macrocells and buried macrocells varies from device to device. The buried macrocell features a register that can be configured as combinatorial, a D flip-flop, a T flip-flop, or a level-triggered latch. Clocking of the register is very flexible. Four global synchronous clocks and a product term clock are available to clock the register. Furthermore, each clock features programmable polarity

so that registers can be triggered on falling as well as rising edges (see the Clocking section). Clock polarity is chosen at the logic block level. The buried macrocell also supports input register capability. Also, the buried macrocell can be configured to act as an input. The *Output Switch Matrix* makes it possible for any microcell output (OR-gate or flip flop) to drive any of the I/O pins connected to the PAL block.

11.3.3 The Lattice Family of CPLDs

The Lattice Semiconductor Corporation, is considered the inventor of in-system programmable (ISP) logic products, and designs, develops and markets a wide range of Field Programmable Gate Arrays (FPGAs), *Field Programmable System Chips* (*FPSCs*) and high-performance ISP programmable logic devices (PLDs). Its product line includes the families of EEPROM-based Lattice pLSI, the ispLSI devices which are basically the same as the pLSI except that they include in-system programmability, the Lattice ispMACH™ 4256V CPLD, and Lattice's 3000, 4000, and 5000 series of CPLDs. The Lattice ispLSI architecture is shown in Figure 11.16.

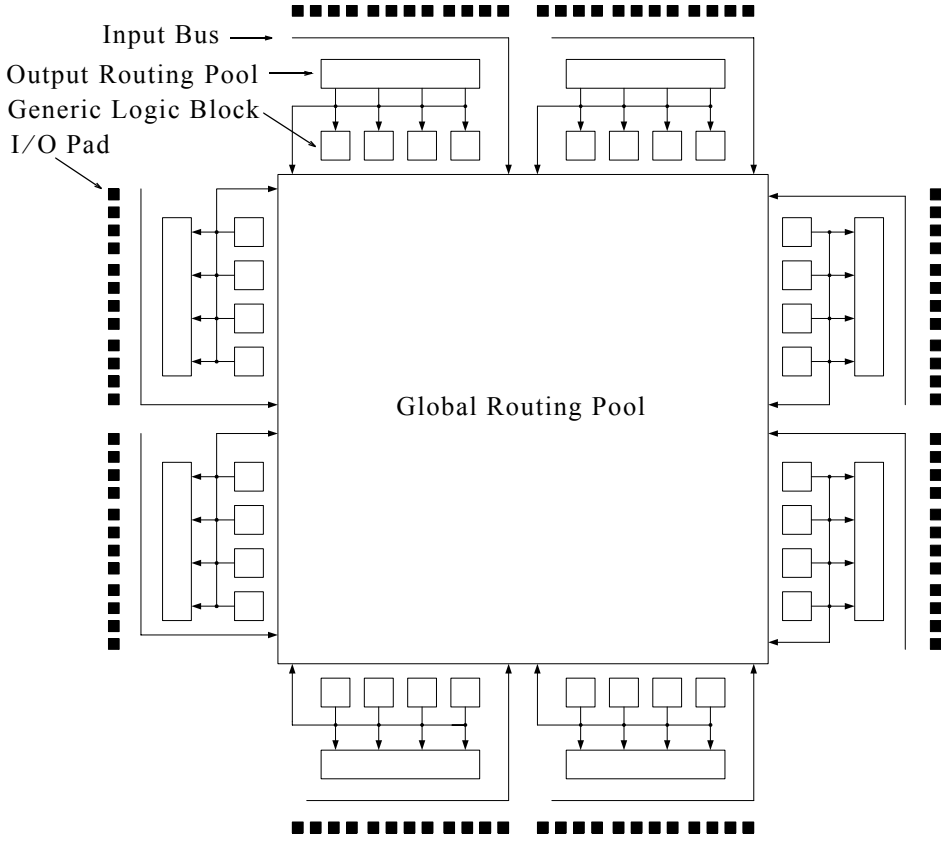


Figure 11.16. Architecture of the Lattice ispLSI CPLD (Courtesy Lattice Semiconductor Corporation)

In Figure 11.16, the I/O Pads are bidirectional devices which are connected to both to the *Global Routing Pool* and the *Generic Logic Blocks*. The global routing pool is a set of wires that intercon-

nect the entire device including the generic logic blocks inputs and outputs. Because all interconnections are routed through the global routing pool, the time delays are predictable as for the AMD Mach devices described in the previous subsection. Each generic logic block each basically a PAL device that consists of an AND plane, a product term allocator, and macrocells as shown in Figure 11.17.

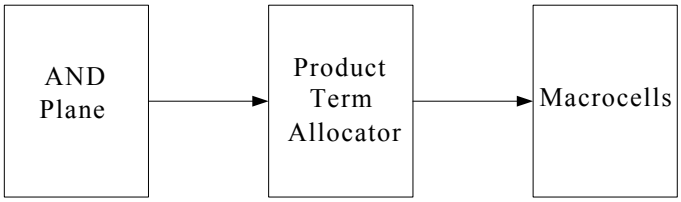


Figure 11.17. Lattice's Generic Logic Block (Courtesy Lattice Semiconductor Corporation)

11.3.4 Cypress FLASH370 Family of CPLDs

The Cypress FLASH370 family CPLDs is based on EEPROM technology, and features a wide variety of densities and pin counts to choose from. At each density there are two packaging options, one that is I/O intensive and the other is register intensive. The CY7C374 and CY7C375 devices of this family both feature 128 macrocells. Figure 11.18 is a block diagram of these devices.

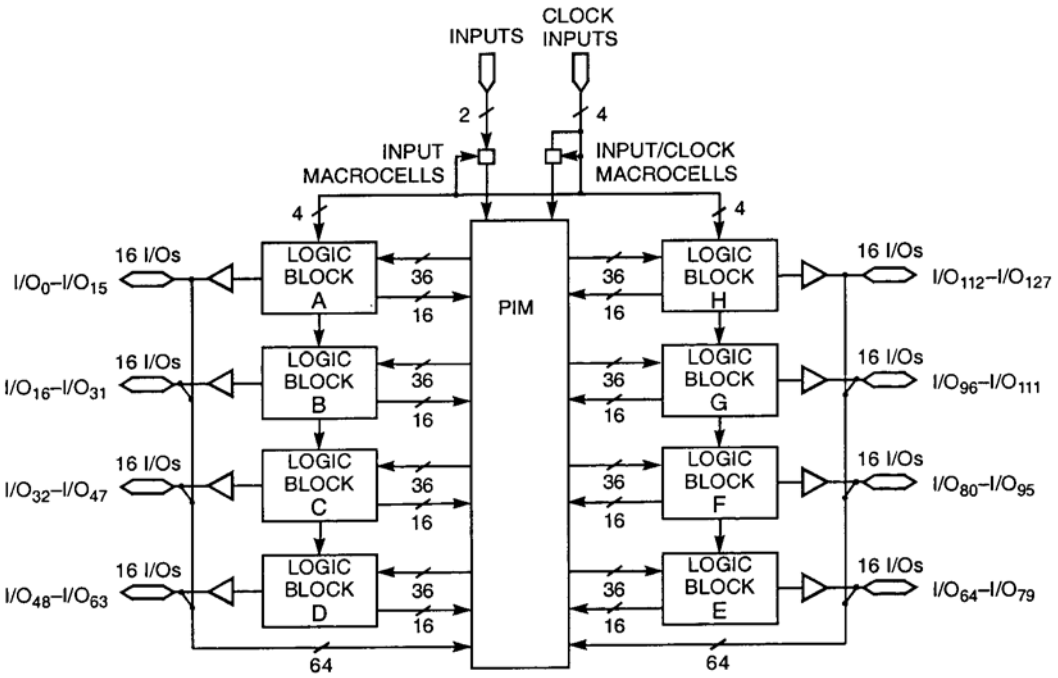


Figure 11.18. Block diagram of the CY7C375 CPLD (Courtesy Cypress Semiconductor Corporation)

On the CY7C374, available in an 84-pin package, half of the macrocells are available on the I/O pins and half are buried, that is, the buried macrocells that do not appear at the I/O pins. On the

CY7C375 all of the macrocells are available to the I/O pins and this device is available as a 160-pin package.

The *Programmable Interconnect Matrix (PIM)* consists of a global routing matrix for signals from I/O pins and feedbacks from the logic blocks. Signals from any pin or any logic block can be routed to any or all logic blocks. Each logic block receives 36 inputs from the PIM and their complements, allowing for 32-bit operations be to implemented in a single pass. All routing is accomplished by software.

The *Logic Block* is the basic building block of the FLASH370 architecture. It consists of a product term array, a product term allocator, 16 macrocells, and a number of I/O cells which varies with the device used. There are two types of logic blocks in the FLASH370 family. The first type, device CY7C375, features an equal number of macrocells and I/O cells as shown in Figure 11.19. This architecture is best suited for I/O intensive applications.

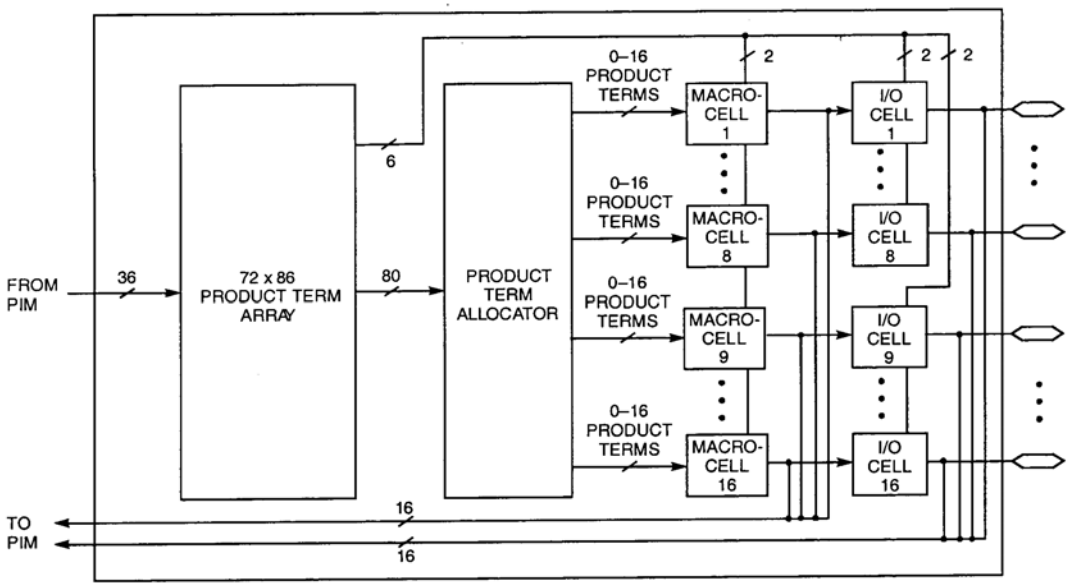


Figure 11.19. Block diagram of the I/O Intensive CY7C375 CPLD (Courtesy Cypress Semiconductor Corp.)

The second type, device CY7C374, of logic block features a buried macrocell along with each I/O macrocell. In other words, there are 8 macrocells that are connected to I/O cells and 8 macrocells that are internally fed back to PIM as shown in Figure 11.20. This architecture is best suited for register-intensive applications.

Each logic block features a 72 by 86 programmable product term array. The array is fed with 36 inputs from the PIM which originate from macrocell feedbacks and device pins. The full 72-input field is generated by both active HIGH and active LOW versions of each of these inputs.

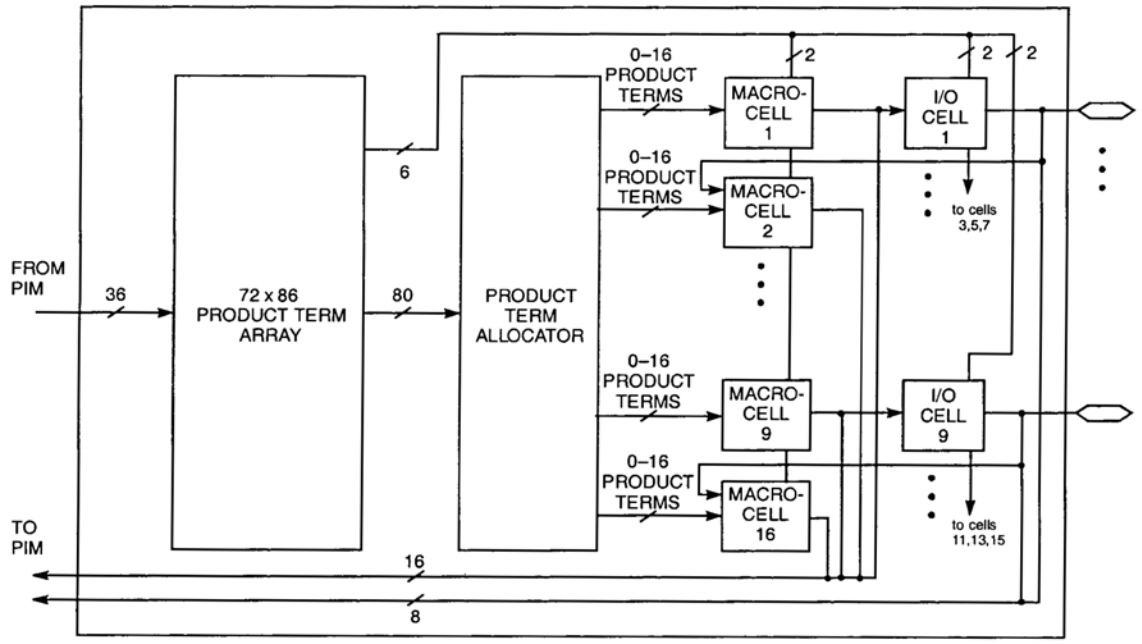


Figure 11.20. Block diagram of the Register Intensive CY7C374 CPLD (Courtesy Cypress Semiconductor Corp.)

Of the 86 product terms, 80 are for general-purpose use for the 16 macrocells in the logic block. Four of the remaining six product terms in the logic block are output enable (OE) product terms. Each of the OE product terms controls up to eight of the 16 macroceIls and is selectable on an individual macrocell basis. In other words, each I/O cell can select between one of two OE product terms to control the output buffer. The first two of these four OE product terms are available to the upper half of the I/O macrocells in a logic block. The other two OE product terms are available to the lower half of the I/O macrocells in a logic block. The final two product terms in each logic block are dedicated asynchronous set and asynchronous reset product terms.

The *Product Term Allocator* serves as a means for the software to automatically distribute product terms among the 16 macrocells in the logic block as needed. A total of 80 product terms are available from the local product term array. The product term allocator provides two important capabilities without affecting performance: product term steering and product term sharing. These are explained below.

Product term steering is the process of assigning product terms to macrocells as needed. For example, if one macrocell requires ten product terms while another needs just three, the product term allocator will "steer" ten product terms to one macrocell and three to the other. On FLASH370 devices, product terms are steered on an individual basis. Any number between 0 and 16 product terms can be steered to any macrocell. Note that 0 product terms is useful in cases where a particular macrocell is unused or used as an input register.

Product term sharing is the process of using the same product term among multiple macrocells. For example, if more than one output has one or more product terms in its equation that are com-

mon to other outputs, those product terms are only programmed once. The FLASH37O product term allocator allows sharing across groups of four output macrocells in a variable fashion. The software automatically takes advantage of this capability – the user needs not to intervene. It should be noted that greater usable density can often be achieved if the user "floats" the pin assignment. This allows the compiler to group macrocells that have common product terms adjacently. Neither product term sharing nor product term steering have any effect on the speed of the product. All worst-case steering and sharing configurations have been incorporated in the timing specifications for the FLASH370 devices. As stated earlier, a FLASH370 macrocell can be either an I/O macrocell or a buried macrocell. These are shown in Figure 11.21.

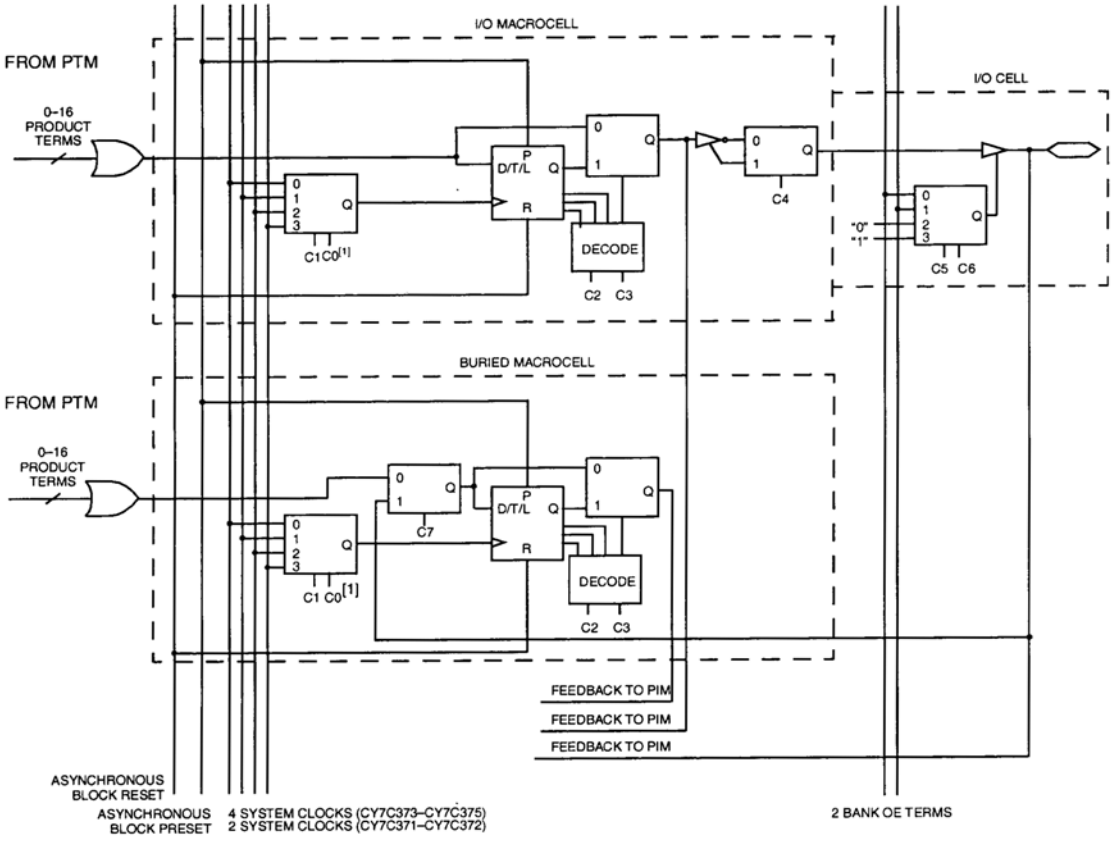


Figure 11.21. I/O and Buried Macrocells in FLASH370 CPLD (Courtesy Cypress Semiconductor Corp.)

Within each logic block there are 8 or 16 I/O macrocells depending on the device used. The macrocell features a register that can be configured as combinational, a D flip-flop, a T flip-flop, or a level-triggered latch. The register can be asynchronously set or asynchronously reset at the logic block level with the separate set and reset product terms. Each of these product terms features programmable polarity. This allows the registers to be set or reset based on an AND expression or an OR expression.

Clocking of the register is very flexible. Depending on the device, either two or four global synchronous clocks are available to clock the register. Furthermore, each clock features programmable polarity so that registers can be triggered on falling as well as rising edges. Clock polarity is chosen at the logic block level. At the output of the macrocell, a polarity control multiplexer is available to select active LOW or active HIGH signals. This has the added advantage of allowing significant logic reduction to occur in many applications. The FLASH370 macrocell features a feedback path to the PIM separate from the I/O pin input path. This means that if the macrocell is buried (fed back internally only), the associated I/O pin can still be used as an input.

The buried macrocell is very similar to the I/O macrocell. It includes a register that can be configured as combinational, a D flip-flop, a T flip-flop, or a latch. The clock for this register has the same options as described for the I/O macro-cell. The primary difference between the I/O macrocell and the buried macrocell is that the buried macrocell does not have the ability to output data directly to an I/O pin. One additional difference on the buried macrocell is the addition of input register capability. The buried macrocell can be configured to act as an input register (D-type or latch) whose input comes from the I/O pin associated with the neighboring macrocell. The output of all buried macrocells is sent directly to the PIM regardless of its configuration.

The I/O cell on the FLASH370 devices is shown on the upper right side in Figure 11.21. The user can program the I/O cell to change the way the three-state output buffer is enabled and/or disabled. Each output can be set permanently on (output only), permanently off (input only), or dynamically controlled by one of two OE product terms.

Six pins on each member of the FLASH370 family are designated as input-only. There are two types of dedicated inputs on FLASH370 devices: input pins and input/clock pins. The architecture for the input pins is shown in Figure 11.22.

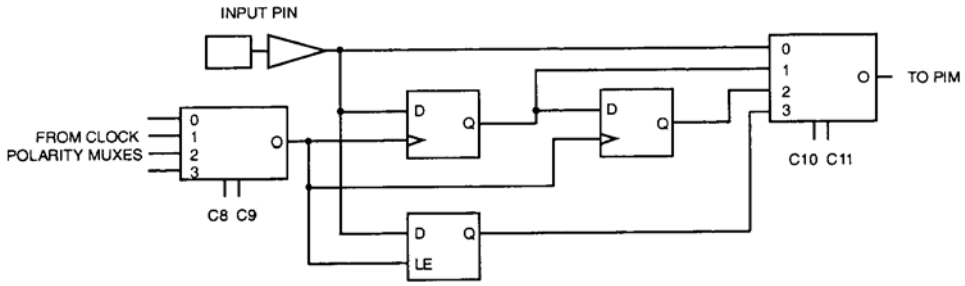


Figure 11.22. Input Pins in FLASH370 CPLD (Courtesy Cypress Semiconductor Corp.)

Four input options are available for the user: combinational, registered, double-registered, or latched. If a registered or latched option is selected, any one of the input clocks can be selected for control.

The architecture for the input/clock pins is shown in Figure 11.23. There are either two or four input/clock pins available, depending on the device selected. Like the input pins, input/clock pins can be combinational, registered, double registered, or latched. In addition, these pins feed the clocking structures throughout the device.

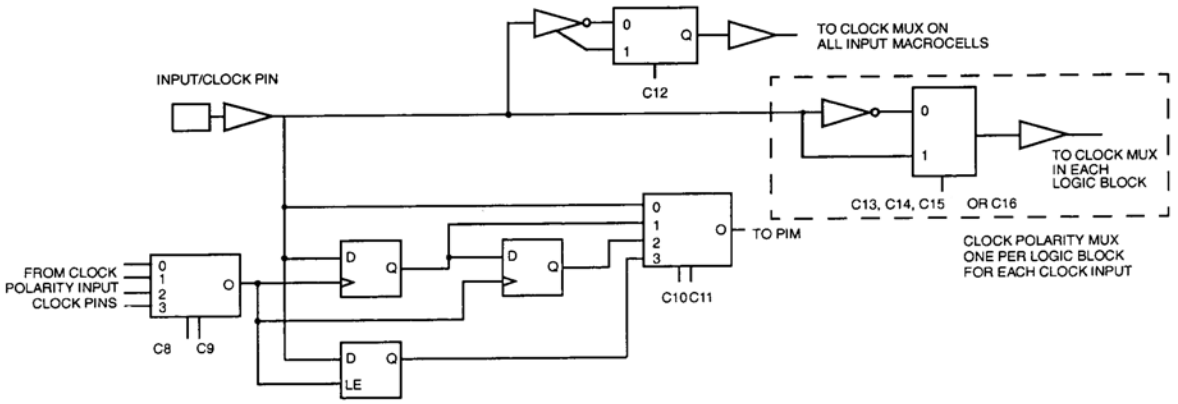


Figure 11.23. Input/Clock Pins in FLASH370 CPLD (Courtesy Cypress Semiconductor Corp.)

The clock path at the input is user-configurable in polarity. The polarity of the clock signal can be controlled by the user and it is separately controlled for input registers and output registers.

One of the most important features of the FLASH370 family is the simplicity of its timing. All delays are worst case and systern performance is unaffected by the features used or not used on the parts. For combinational paths, any input to any output incurs an 8.5 ns worst-case delay regardless of the input/clock pins while the other devices have four input/clock amount of logic used. For synchronous systems, the input set-up time to the output macrocells for any input is 5.0 ns and the clock to output time is 6.0 ns. These times are for any output and clock, regardless of the logic used.

The FLASH370 family of CPLDs is supported by third-party design software vendors including ABEL.*

11.3.5 Xilinx XC9500 Family of CPLDs

The XC9500 series is the latest family of Xilinx's in-system programmable family of CPLDs. All devices in this family are designed for a minimum of 10,000 program/erase cycles, and include IEEE 1149.1 (JTAG) boundary-scan† support. The architecture of the XC9500 is shown in Figure 11.24. The logic density of the XC9500 devices ranges from 800 to over 6,400 usable gates with 36 to 288 registers, respectively. The XC9500 family is fully pin-compatible allowing easy design migration across multiple density options in a given package footprint.

The XC9500 architectural features address the requirements of in-system programmability. Enhanced pin-locking capability avoids costly board rework. An expanded JTAG instruction set allows version control of programming patterns and in-system debugging.

* *ABEL is a trademark of Data I/O Corp. An introduction to ABEL is presented in Appendix A.*
† *The IEEE 1149.1 is discussed in Appendix B.*

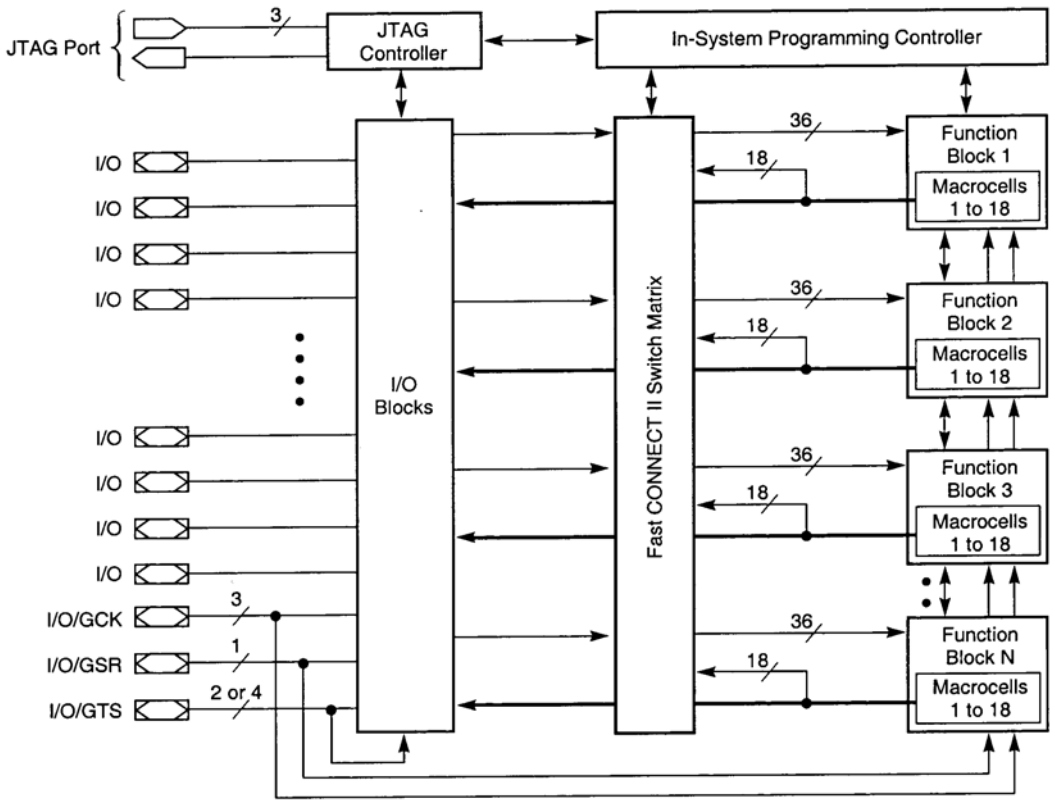


Figure 11.24. .Architecture of the XC9500 CPLD (Courtesy Xilinx, Inc.)

Each Function Block, shown in Figure 11.25, is comprised of 18 independent macrocells, each capable of implementing a combinational or registered function. The function block also receives global clock, output enable, and set/reset signals. It generates 18 outputs that drive the Fast CONNECT switch matrix. These 18 outputs and their corresponding output enable signals also drive the I/O blocks.

Logic within each function block is implemented using a sum-of-products representation. Thirty-six inputs provide 72 true and complement signals into the programmable AND-array to form 90 product terms. Any number of these product terms, up to the 90 available, can be allocated to each macrocell by the product term allocator. Each function block (except for the XC9536) supports local feedback paths that allow any number of function block outputs to drive into its own programmable AND-array without going outside the function block. These paths are used for creating very fast counters and state machines where all state registers are within the same function block.

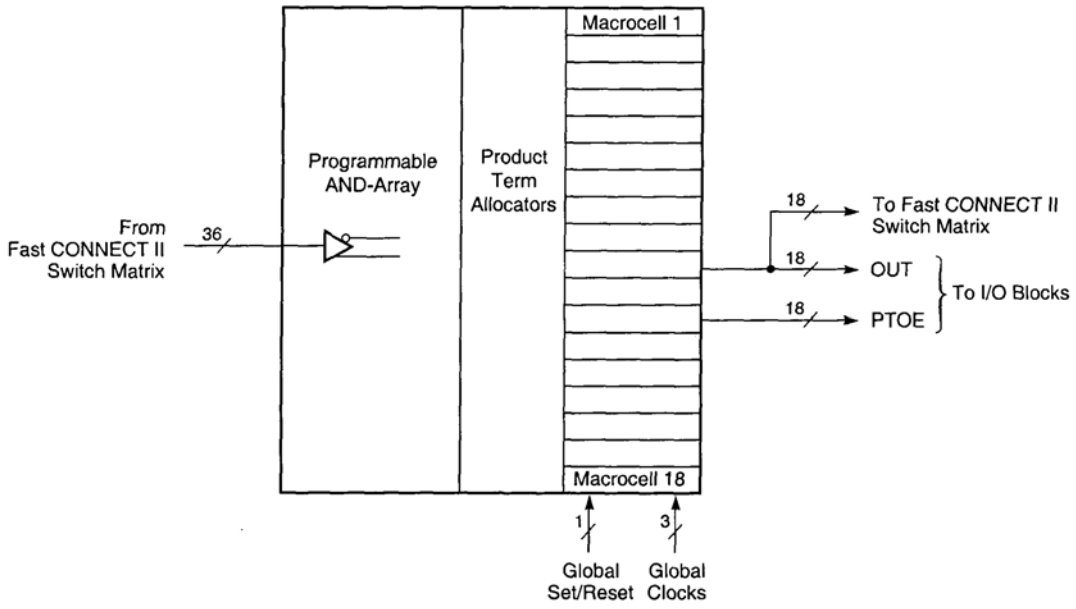


Figure 11.25. The XC9500 CPLD Function Block (Courtesy Xilinx, Inc.)

Each XC9500 macrocell may be individually configured for a combinational or registered function. The macrocell and associated function block logic is shown in Figure 11.26. Five direct product terms from the AND-array are available for use as primary data inputs (to the OR and XOR gates) to implement combinational functions, or as control inputs including clock, set/reset, and output enable. The product term allocator associated with each macrocell selects how the five direct terms are used.

The macrocell register carl be configured as a D-type or T-type flip-flop, or it may be bypassed for combinational operation. Each register supports both asynchronous set and reset operations. During power-up, all user registers are initialized to the user-defined preload state (default to 0 if unspecified).

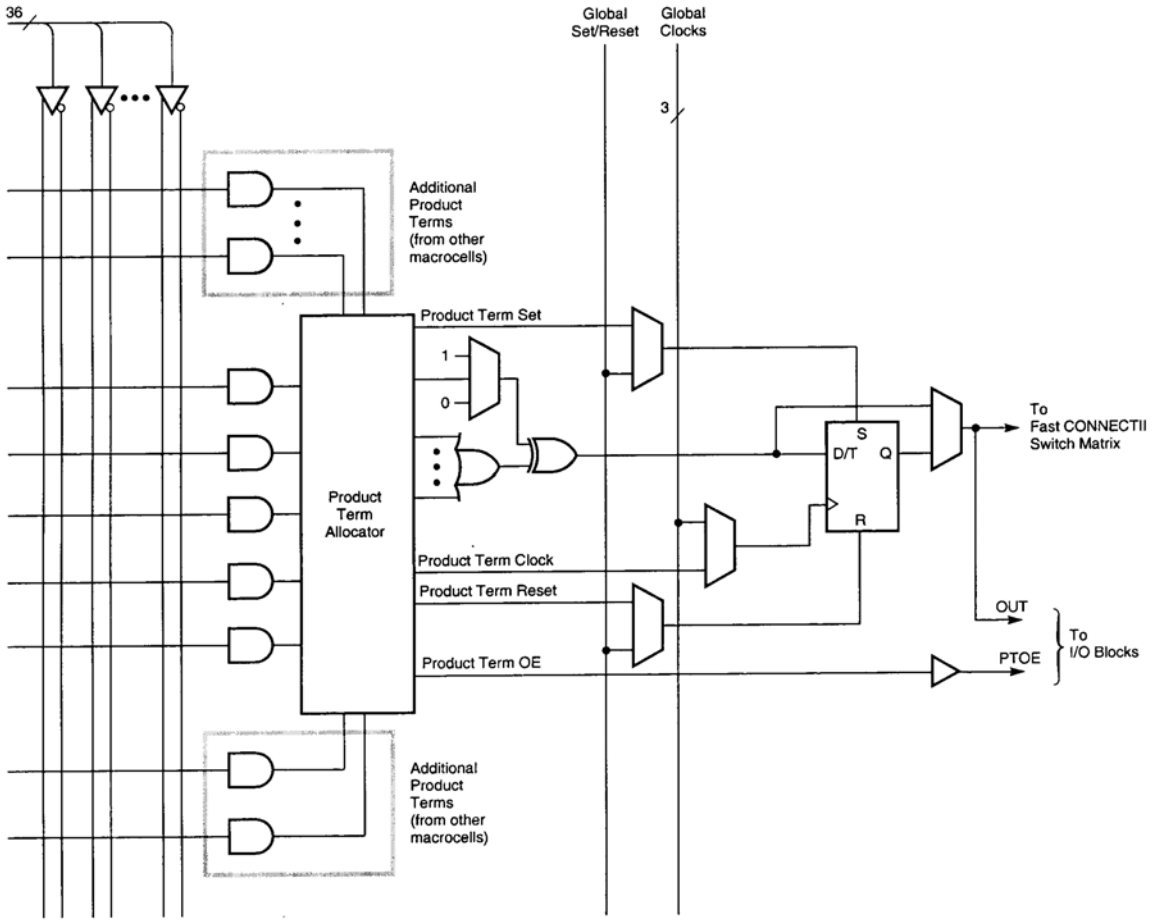


Figure 11.26. XC9500 Macrocell Inside a Function Block (Courtesy Xilinx, Inc.)

All global control signals are available to each individual macrocell, including clock, set/reset, and output enable signals. As shown in Figure 11.27, the macrocell register clock originates from either of three global clocks or a product term clock. Both true and complement polarities of a Global Clock pin can be used within the device. A Global Set/Reset input is also provided to allow user registers to be set to a user-defined state.

The product term allocator determines how the five direct product terms are assigned to each macrocell. For instance, all five direct terms can drive the OR function as shown in Figure 11.28.

The product term allocator can re-assign other product terms within the function block to increase the logic capacity of a macrocell beyond five direct terms. Any macrocell requiring additional product terms can access uncommitted product terms in other macrocells within the function block. Up to 15 product terms can be available to a single macrocell as shown in Figure 11.29.

The incremental delay affects only the product terms in other macrocells. The timing of the direct product terms is not changed.

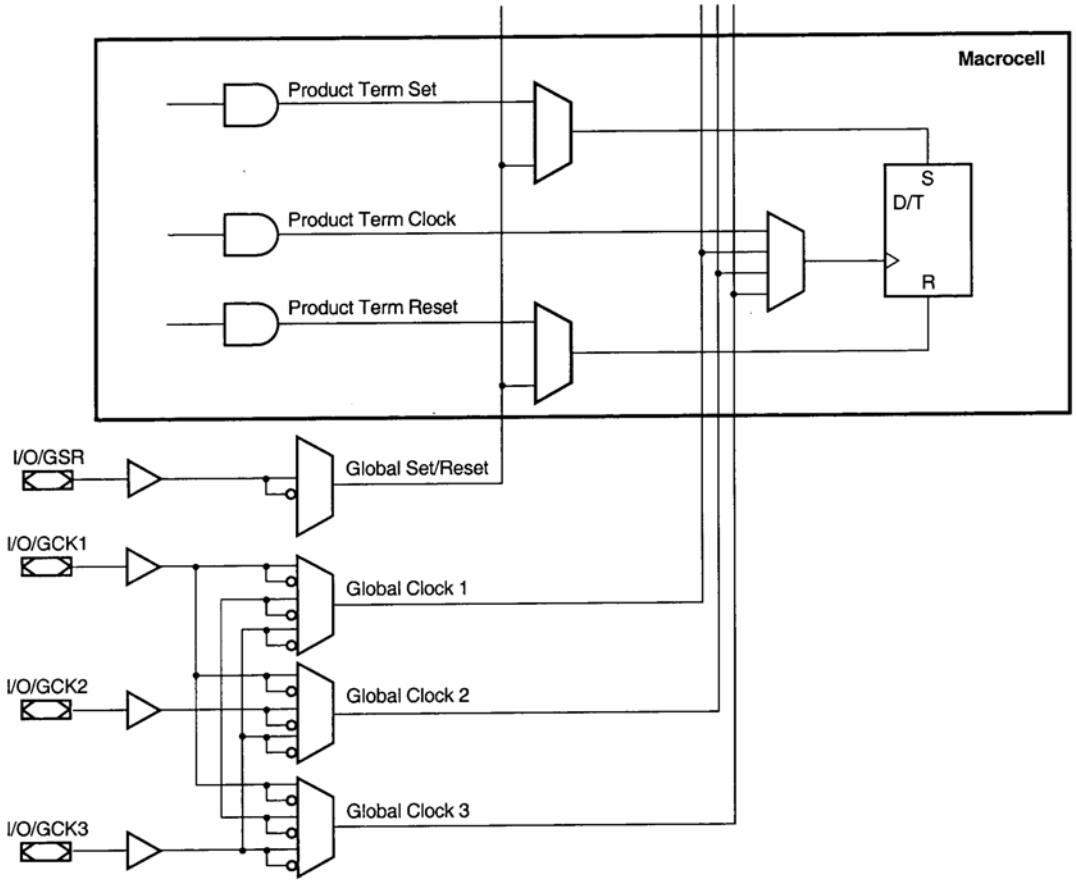


Figure 11.27. XC9500 Macrocell Clock and Set/Reset Capability (Courtesy Xilinx, Inc.)

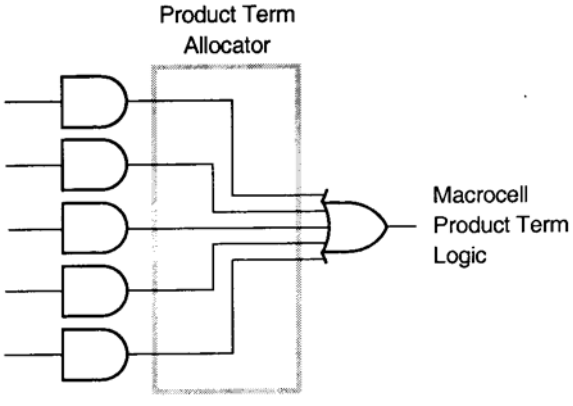


Figure 11.28. XC9500 Macrocell Logic Using Direct Product Term (Courtesy Xilinx, Inc.)

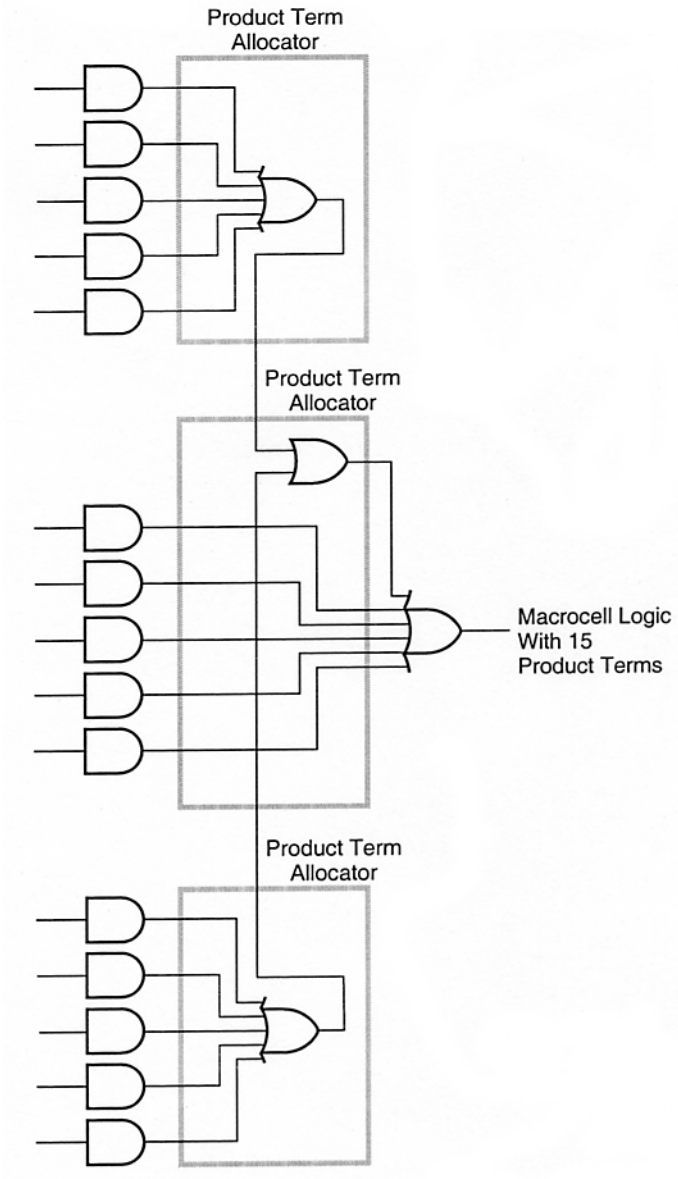


Figure 11.29. XC9500 Product Term Allocation with 15 Product Terms (Courtesy Xilinx, Inc.)

The product term allocator can re-assign product terms from any macrocell within the function block by combining partial sums of products over several macrocells as shown in Figure 11.30. All 90 product terms are available to any macrocell.

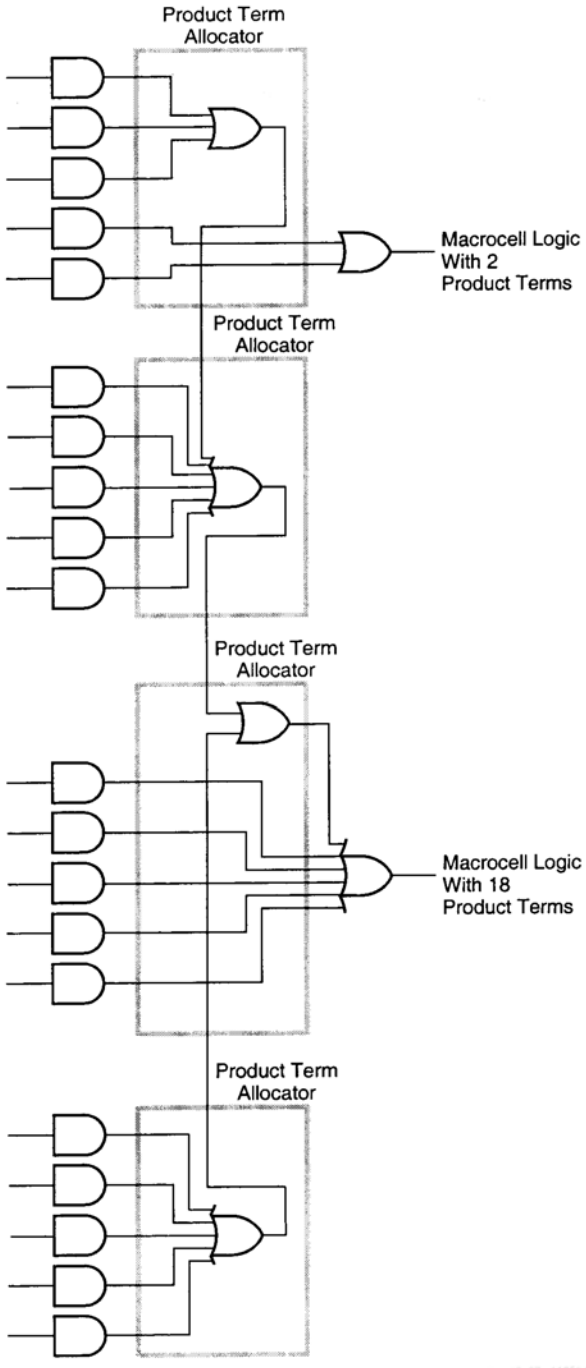


Figure 11.30. XC9500 Product Term Allocation over Several Macrocells (Courtesy Xilinx, Inc.)

The internal logic of the product term allocator is shown in Figure 11.31.

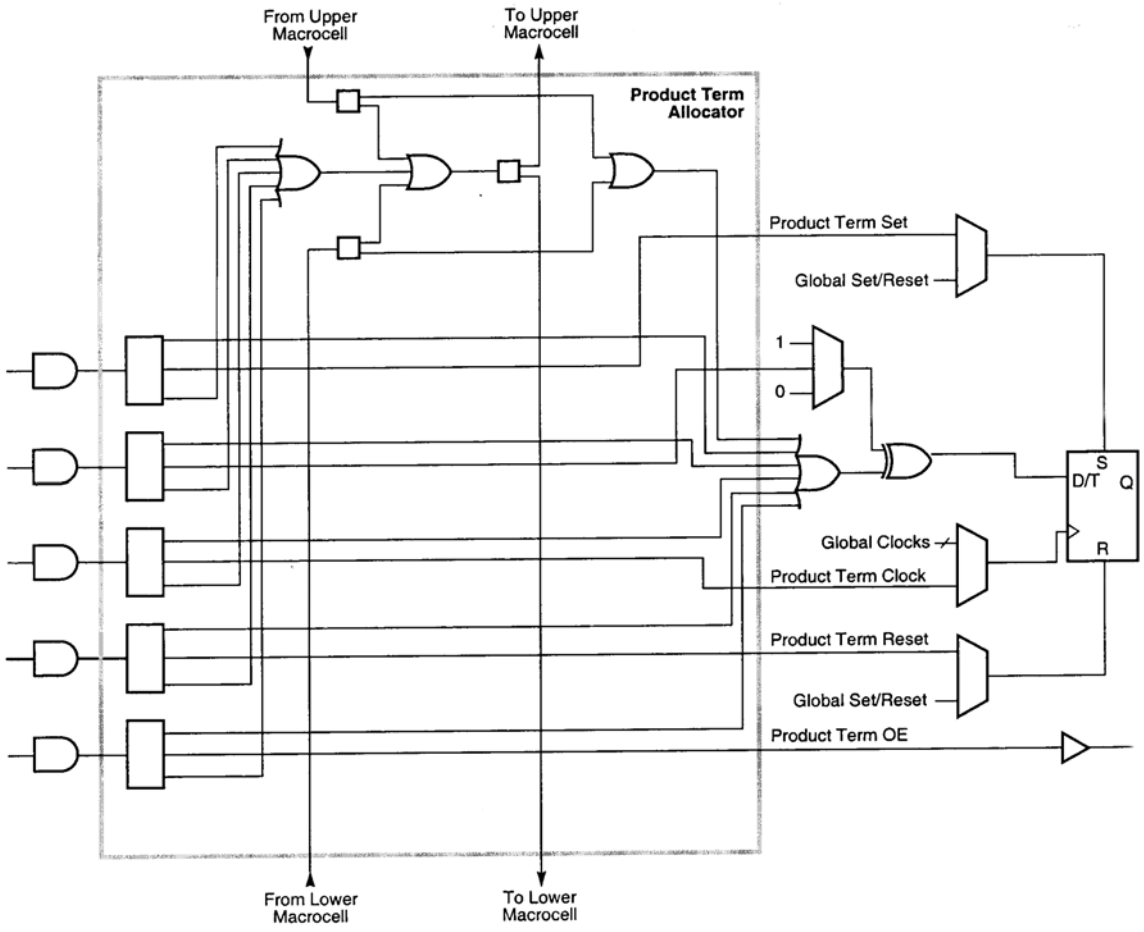


Figure 11.31. XC9500 Product Term Allocator Logic (Courtesy Xilinx, Inc.)

The Fast CONNECT switch matrix connects signals to the function block inputs, as shown in Figure 11.32. All I/O block outputs (corresponding to user pin inputs) and all function block outputs drive the Fast CONNECT matrix. Any of these (up to a function block fan-in limit of 36) may be selected, through user programming, to drive each function block with a uniform delay.

The Fast CONNECT switch matrix is capable of combining multiple internal connections into a single wired-AND output before driving the destination function block. This provides additional logic capability and increases the effective logic fan-in of the destination function block without any additional timing delay. This capability is available for internal connections originating from function block outputs only. It is automatically invoked by the development software where applicable.

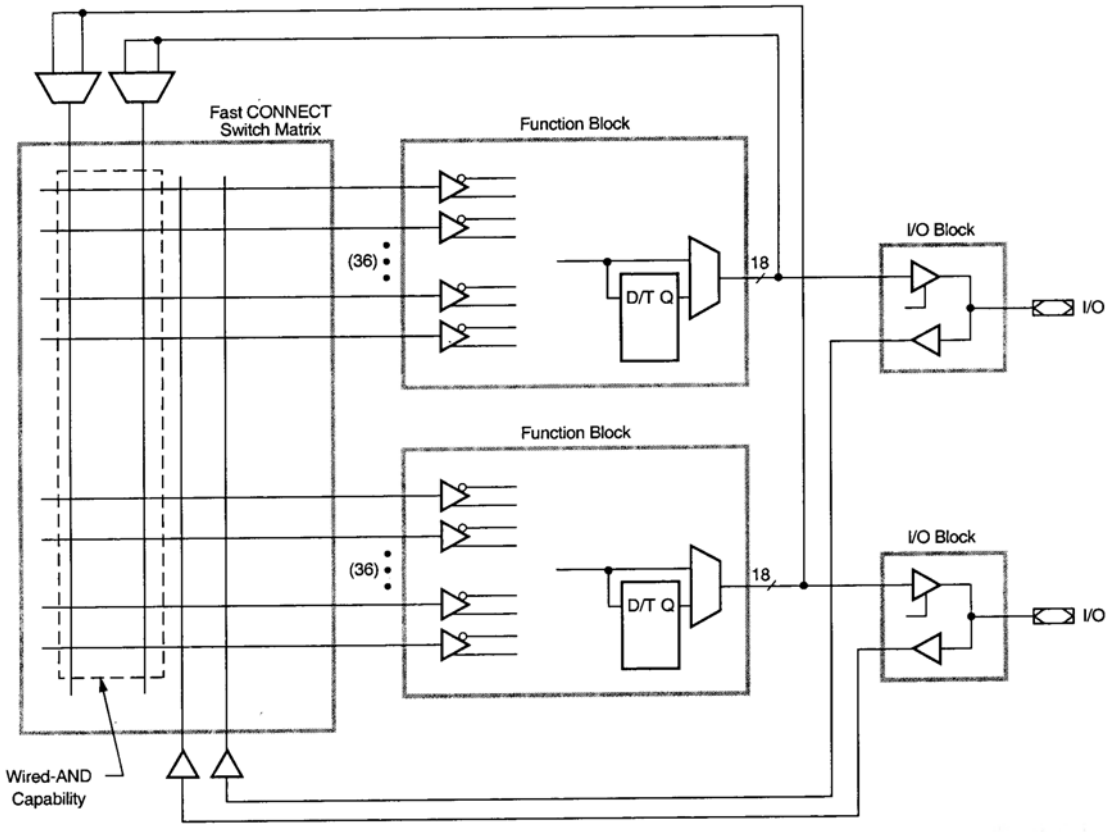


Figure 11.32. The XC9500 CPLD Fast CONNECT Switch Matrix (Courtesy Xilinx, Inc.)

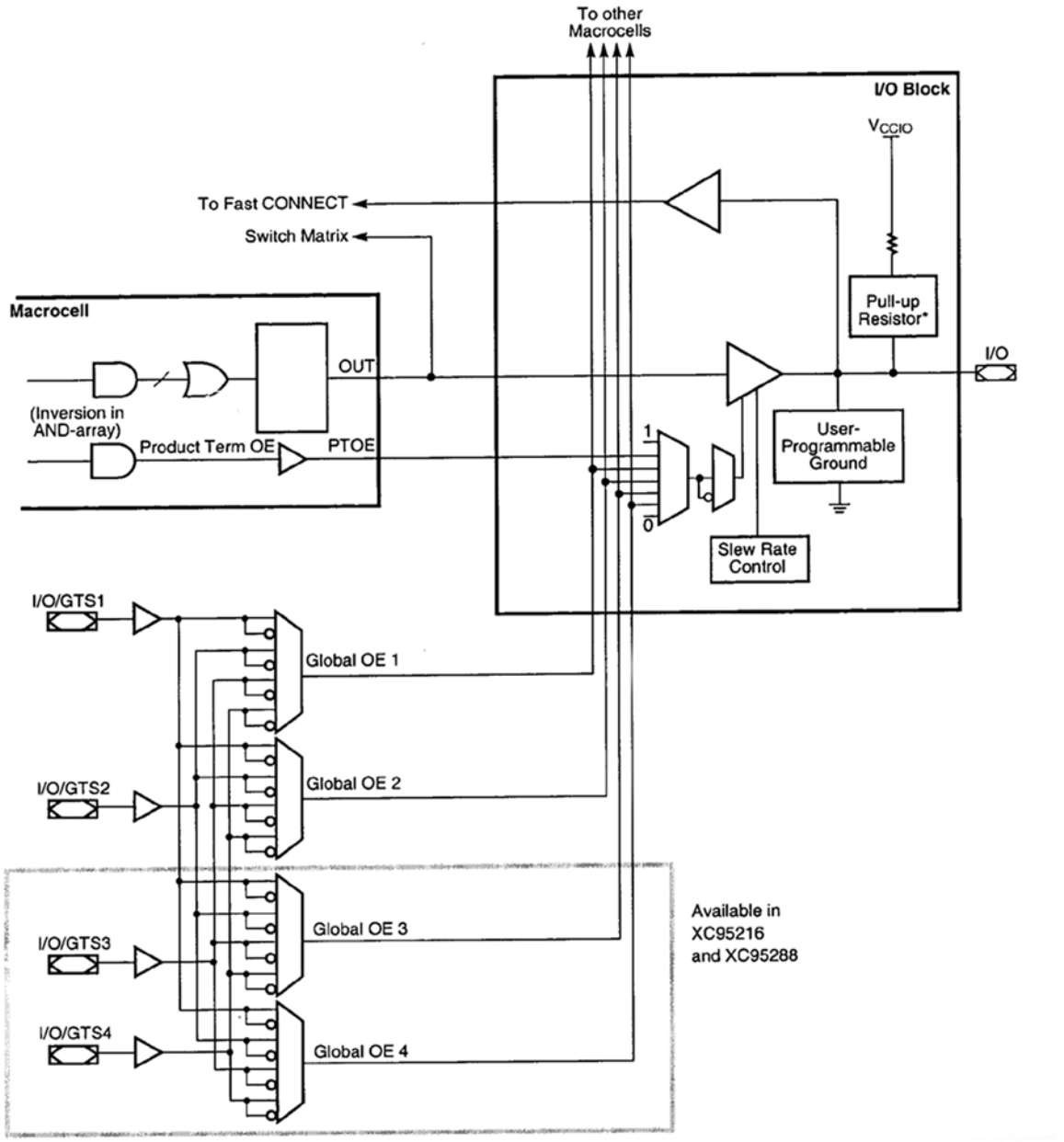


Figure 11.33. The XC9500 CPLD I/O Block and Output Enable Capability (Courtesy Xilinx, Inc.)

XC9500 devices are programmed in-system via a standard 4-pin JTAG protocol, as shown in Figure 11.34. In-system programming offers quick and efficient design iterations and eliminates package handling. The Xilinx development system provides the programming data sequence using a Xilinx download cable, a third-party JTAG development system, JTAG-compatible board tester, or a simple microprocessor interface that emulates the JTAG instruction sequence. All I/Os are 3-stated and pulled high by the lOB resistors during in-system programming. If a particular signal must remain Low during this time, then a pull-down resistor may be added to the pin.

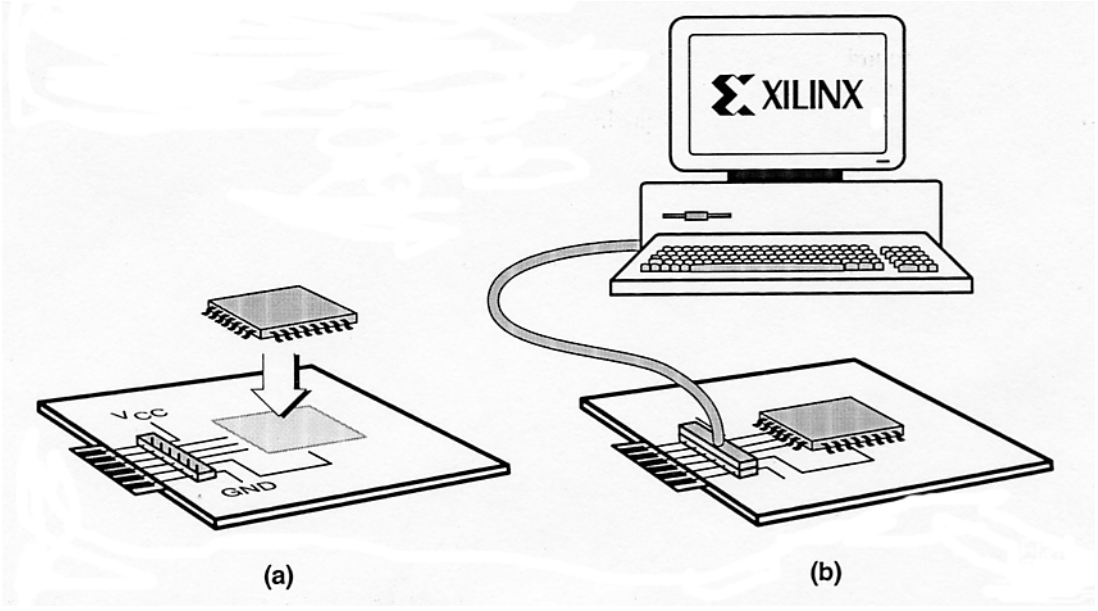


Figure 11.34. The XC9500 CPLD In-System Programming Operation: (a) Solder Device to PCB and (b) Program Using Download Cable (Courtesy Xilinx, Inc.)

XC9500 devices can also be programmed by the Xilinx HW13O device programmer as well as third-party programmers. This provides the added flexibility of using pre-programmed devices during manufacturing, with an in-system programmable option for future enhancements.

11.3.6 CPLD Applications

CPLDs are used as Local Area Network (LAN) controllers, cache memory controllers, graphics controllers, Universal Asynchronous Receiver/Transmitters (UARTs), and in general in applications that require large quantities of AND/OR gates but not a large number of flip flops. With in-system programmable CPLDs it is possible to reconfigure hardware without powering-down the system.

Example 11.3

Implement a 24-bit adder with a 168-input CPLD using:

a. 1-bit stages

b. 2-bit stages

c. 3-bit stages

Use the ABEL High Level Design Language (HLD) to implement the equations, and assume that each macrocell in the CPLD contains just one XOR gate.

Solution:

An introduction to ABEL is presented in Appendix A. For this example, we will be using the logic operators AND, OR, XOR, and NOT (Inversion) extensively. For convenience, they are listed in Table 11.2 below.

TABLE 11.2

Logic Operator	ABEL Symbol
AND	&
OR	#
XOR	$
NOT	!

From Chapter 7, after replacing X (augend) and Y (addend) with A and B respectively, the sum S and present carry C_{OUT} in terms of A, B, and previous carry C_{IN}, were derived as

$$\begin{aligned} S &= \overline{A}\overline{B}C_{IN} + \overline{A}B\overline{C}_{IN} + A\overline{B}\overline{C}_{IN} + ABC_{IN} = (AB + \overline{A}\overline{B})C_{IN} + (\overline{A}B + A\overline{B})\overline{C}_{IN} \\ &= \overline{(A \oplus B)}C_{IN} + (A \oplus B)\overline{C}_{IN} = A \oplus B \oplus C_{IN} \end{aligned} \tag{11.1}$$

and

$$C_{OUT} = AB + AC_{IN} + BC_{IN} = AB + (A + B)C_{IN} \tag{11.2}$$

For high-speed addition, in Chapter 10 we found that a carry output C_{i+1} is generated whenever the logical expression (11.3) below is true.

$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i = A_iB_i + (A_i + B_i)C_i \tag{11.3}$$

and letting

$$\begin{aligned} G_i &= A_iB_i \\ P_i &= A_i + B_i \end{aligned} \tag{11.4}$$

Substitution of (11.4) into (11.3) yields

$$C_{i+1} = G_i + P_iC_i \tag{11.5}$$

and in general

$$\begin{aligned} C_{i+1} &= G_i + P_iC_i = G_i + P_i(G_{i-1} + P_{i-1}C_{i-1}) = G_i + P_iG_{i-1} + P_iG_{i-1} + P_iP_{i-1}C_{i-1} \\ &= G_i + P_iG_{i-1} + P_iP_{i-1}(G_{i-2} + P_{i-2}C_{i-2}) \\ &= G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + P_iP_{i-1}P_{i-2}C_{i-2} \\ &\quad \dots \quad \dots \quad \dots \quad \dots \quad \dots \quad \dots \\ &= G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \dots + P_iP_{i-1}P_{i-2}\dots P_1P_0C_0 \end{aligned} \tag{11.6}$$

a. From 11.5 or 11.6,

$$C_{IN1} = C_{OUT0} = G_0 + P_0C_{IN0}$$
$$\texttt{CIN1 = COUT0 = G0\#(P0\&CIN0)} \tag{11.7}$$

where the first line in (11.7) is the Boolean expression, and the second is the equivalent expression in ABEL. We will follow this practice throughout this example. Likewise,

$$C_{IN2} = C_{OUT1} = G_1 + P_1C_{OUT0} = G_1 + P_1(G_0 + P_0C_{IN0}) = G_1 + P_1G_0 + P_1P_0C_{IN0}$$
$$\texttt{CIN2 = COUT1 = G1\#(P1\&COUT0) = G1\#P1\&G0\#} \text{ (P1\& P0 \& CIN0)} \tag{11.8}$$

$$\begin{aligned} C_{IN3} &= C_{OUT2} = G_2 + P_2C_{OUT1} = G_2 + P_2(G_1 + P_1G_0 + P_2P_1P_0C_{IN0}) \\ &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_{IN0} \end{aligned}$$
$$\begin{aligned} \texttt{CIN3} &= \texttt{COUT2 = G2 \# (P2 \& COUT1)} \\ &= \texttt{G2 \# (P2 \& G1) \#} \text{ (P2 \& P1 \& G0) \# (P2 \& P1\& P0 \& CIN0)} \end{aligned} \tag{11.9}$$

$$\begin{aligned} C_{IN4} &= C_{OUT3} = G_3 + P_3C_{OUT2} = G_3 + P_3(G_2 + P_2G_1 + P_2P_1P_0C_{IN0}) \\ &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_{IN0} \end{aligned}$$
$$\begin{aligned} \texttt{CIN4} &= \texttt{COUT3 = G3 \# (P3 \& COUT2)} \\ &= \texttt{G3 \# (P3 \& G2) \#} \text{ (P3 \& P2 \& G1) \# (P3 \& P2\& P1 \& G0)} \\ &\quad \text{\# (P3 \& P2 \& P1 \& P0 \& CIN0)} \end{aligned} \tag{11.10}$$

Figure 11.35 shows how each stage of the multiple-bit adder can compute the sum by substituting the C_{OUT} developed from the generate and propagate outputs of the previous stage for the C_{IN} using the expression

$$S = A \oplus B \oplus C_{IN}$$
$$\text{SUM = A \$ B \$ CIN} \tag{11.11}$$

In Figure 11.35, there are three levels of logic. At the first level, each stage computes P and G. The second level computes C_{OUT} from each stage for use as the C_{IN} for the next higher order stage. At the third level, each stage of the adder computes the sum of A, B, and C_{IN} as in relation (11.11) above.

b. A close examination of relations (11.7) through (11.10) reveals that each equation for C_{OUT} uses one more product term at each stage than the previous stage used. Since a typical CPLD has a limited number of product terms per macrocell, an excessive number of product terms in large adders would create a problem. An option would be to break the equations for C_{OUT} into two levels, but this would create a considerable delay. A better option is to use adder stages that consist of two-bit full adders instead of a one-bit full adders as in part (a). Figure 11.36 shows a 24-bit adder using two-bit stages.

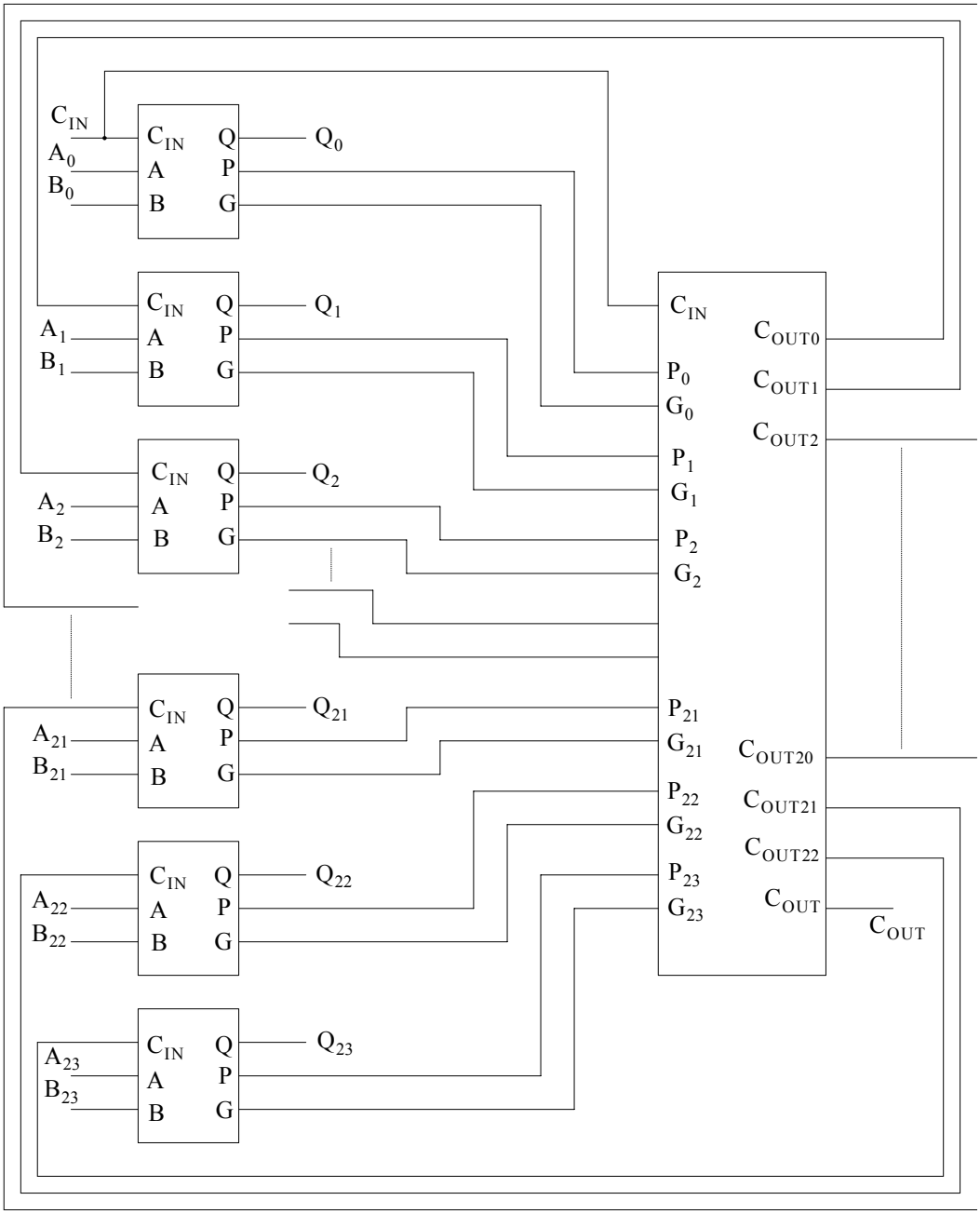


Figure 11.35. 24-bit adder using one-bit stages

In the adder of Figure 11.36, the propagate and generate pairs are formed from each two-bit stage to form the carry to the next higher two-bit stage. Also, using two-bit adders we only need half as many stages, and this results in half as many propagate and generate pairs from stage to stage for a given size adder. Accordingly, the highest order C_{IN} equation requires half as many product terms.

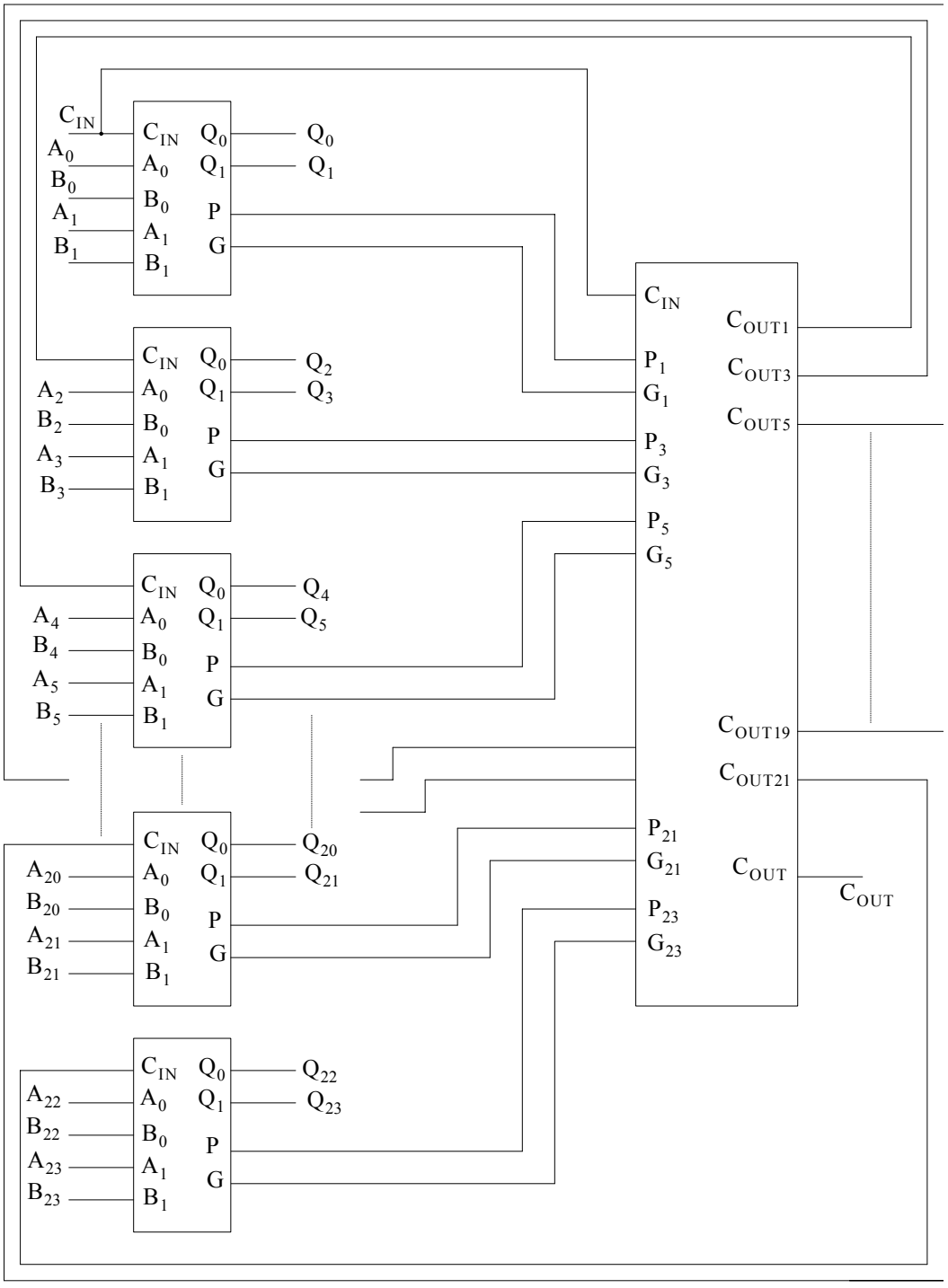


Figure 11.36. 24-bit adder using two-bit Stages

The sum equations for the two-bit adder of Figure 11.36 are as follows:

$$S_0 = A_0 \oplus B_0 \oplus C_{IN0}$$
$$\text{SUM0} = \text{A0 \$ B0 \$ CIN0} \tag{11.12}$$

Implementation of the expression in (11.12) requires two XOR gates, and since we are told that there is only a XOR gate in a macrocell, we will use it and we will expand the other, that is, we express (11.12) as

$$S_0 = A_0 \oplus (B_0 \oplus C_{IN0}) = A_0 \oplus (B_0\overline{C}_{IN0} + \overline{B}_0 C_{IN0})$$
$$\text{SUM0} = \text{A0 \$ ((B0 \& !CIN0) \# (!B0 \& CIN0))} \tag{11.13}$$

Likewise,

$$S_1 = A_1 \oplus (B_1\overline{C}_{IN1} + \overline{B}_1 C_{IN1})$$
$$\text{SUM1} = \text{A1 \$ ((B1 \& !CIN1) \# (!B1 \& CIN1))} \tag{11.14}$$

Also,

$$C_{IN1} = C_{OUT0} = A_0B_0 + A_0C_{IN0} + B_0C_{IN0} = A_0(B_0 + C_{IN0}) + B_0C_{IN0}$$
$$\text{CIN1} = \text{COUT0} = \text{(A0 \& B0) \# (A0 \& CIN0) \# (B0 \& CIN0)} \tag{11.15}$$

Substitution of (11.15) into (11.14) yields

$$\begin{aligned} S_1 &= A_1 \oplus [B_1\overline{(A_0B_0 + A_0C_{IN0} + B_0C_{IN0})} + \overline{B}_1(A_0B_0 + A_0C_{IN0} + B_0C_{IN0})] \\ &= A_1 \oplus [B_1(\overline{A}_0 + \overline{B}_0) \cdot B_1(\overline{A}_0 + \overline{C}_{IN0}) \cdot B_1(\overline{B}_0 + \overline{C}_{IN0}) + \overline{B}_1(A_0B_0 + A_0C_{IN0} + B_0C_{IN0})] \\ &= A_1 \oplus [(\overline{A}_0B_1 + \overline{B}_0B_1) \cdot (\overline{A}_0B_1 + B_1\overline{C}_{IN0}) \cdot (\overline{B}_0B_1 + B_1\overline{C}_{IN0}) + \overline{B}_1(A_0B_0 + A_0C_{IN0} + B_0C_{IN0})] \end{aligned}$$

We observe that

$$\begin{aligned} \overline{A}_0B_1 \cdot \overline{A}_0B_1 &= \overline{A}_0B_1 \\ \overline{A}_0B_1 \cdot B_1\overline{C}_{IN0} &= \overline{A}_0B_1\overline{C}_{IN0} \\ \overline{A}_0B_1 \cdot \overline{B}_0B_1 &= \overline{A}_0\overline{B}_0B_1 \\ \overline{B}_0B_1 \cdot B_1\overline{C}_{IN0} &= \overline{B}_0B_1\overline{C}_{IN0} \\ \overline{B}_0B_1 \cdot \overline{B}_0B_1 &= \overline{B}_0B_1 \end{aligned} \tag{11.16}$$

From the above relation we see that A_1 must be XORed with the quantity in brackets. Since the sum S_1 can be either logical 0 or logical 1, if $A_1 = 0$, the first 4 terms on the right side of (11.16) must be selected such that the sum will be logical 1. The first and last, i.e, $\overline{A}_0B_1$, and $\overline{B}_0B_1$ do not include the carry C_{IN0} and thus they are ignored. From the second, third and fourth equations in 11.16, we see that for the sum to be logical 1 when $A_1 = 0$, the variable B_1 must be logical 1 while $A_0 = 0$ and $C_{IN0} = 0$, or $B_0 = 0$ and $C_{IN0} = 0$. If $A_1 = 1$ the terms $A_0B_0\overline{B}_1$, $A_0\overline{B}_1C_{IN0}$, and $B_0\overline{B}_1C_{IN0}$ must be zero so that the sum will be logical 1. Our sum equation S_1 then reduces to

$$S_1 = A_1 \oplus [(\overline{A}_0\overline{B}_0B_1 + \overline{A}_0B_1\overline{C}_{IN0} + \overline{B}_0B_1\overline{C}_{IN0}) + (A_0B_0\overline{B}_1 + A_0\overline{B}_1C_{IN0} + B_0\overline{B}_1C_{IN0})]$$

```
SUM1 = A1 $ ((!A0 & !B0 & B1) # (!A0 & B1 & !CIN0) # (!B0 & B1 & !CIN0)
          # (A0 & B0 & !B1) # (A0 & !B1 & CIN0) # (B0 & !B1 & CIN0))
```

(11.17)

The generated equations for the two-bit adder of Figure 11.36 are:

$$\begin{aligned} G_0 &= A_0B_0 \\ G_1 &= A_1B_1 + A_0B_0A_1 + A_0B_0B_1 \end{aligned}$$

```
G0 = A0 & B0
G1 = A1 & B1 # A0 & B0 & A1 # A0 & B0 & B1
```

(11.18)

As shown in (11.18), G_1 is generated either when both A_1 and B_1 are logical 1 or when at least one of the addends in the second stage is logical 1 and there is also a carry being generated within the first stage of the two-bit adder.

For the propagated equation, we can obtain a simpler expression if we consider the case where P_1 is logical 0. This occurs when both A and B are logical 0 in either stage of the two-bit adder. Thus,

$$\overline{P}_1 = \overline{(A_0 + B_0) \cdot (A_1 + B_1)} = \overline{A}_0\overline{B}_0 + \overline{A}_1\overline{B}_1$$

```
!P1 = (!A0 & !B0) #  (!A1 & !B1)
```

(11.19)

Although the sum equations and the propagate-generate equations are more complex for the two-bit full adder than for the one-bit adder, there are less product terms needed for the carry-outs C_{OUT} since there is only one propagate-generate pair for every two bits being summed.

The C_{IN} and C_{OUT} equations are the same when using two-bit adders as when using one-bit adders, except when using two-bit adders C_{IN} is the carry-in and C_{OUT} is the carry-out for the two-bit adder stage. Therefore, for a given size adder, using two-bit adders involves half as many stages, half as many carries and propagate/generate pairs, and half as many product terms for the highest order equation for the carry-in, C_{IN}.

11.4 Field Programmable Gate Arrays (FPGAs)

A *Field Programmable Gate Array* (FPGA) is similar to a PLD, but whereas PLDs are generally limited to hundreds of gates, FPGAs support thousands of gates. They are especially popular for prototyping integrated circuit designs. Once the design is set, hardwired chips are produced for faster performance.

Field Programmable Gate Arrays (FPGAs) are divided into two major categories:

1. SRAM-based FPGAs

2. Antifuse-based* FPGAs

11.4.1 SRAM-Based FPGA Architecture

There is a wide range of FPGAs provided by many semiconductor vendors including Xilinx, Altera, Atmel, and Lattice. Each manufacturer provides each own unique architecture. A typical FPGA consists of an array of logic elements and programmable routing resources used to provide the connectivity between the logic elements, FPGA I/O pins, and other resources such as on-chip memory. The structure and complexity of the logic elements, as well as the organization and functionality supported by the interconnection hierarchy, is what distinguishes the different devices from each other. Other features such as block memory and delay-locked-loop technology are also significant factors that influence the complexity and performance of an algorithm implemented using FPGAs. A *logic element* usually consists of one or more RAM-based *n*-input *look-up tables* (LUTs) where *n* is a number between three and six, and one or more flip-flops. LUTs are used to implement combinational logic. A typical logic element is shown in Figure 11.37.

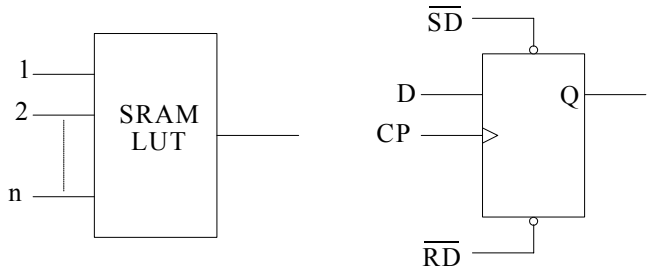


Figure 11.37. Typical logic element in a SRAM-based FPGA

There may also be additional hardware support in each logic element to enable other high-speed arithmetic and logic operations. A typical SRAM-based FPGA architecture is shown in Figure 11.38. The bold lines in Figure 11.38 indicate how connections among two or more logic elements and I/O ports can be made.

11.4.1.1 Xilinx FPGAs

The Xilinx Virtex series of SRAM-based FPGAs are very popular FPGAs. The logic elements, referred to as *slices*, comprise 2 four-input look-up tables (LUTs), 2 flip flops, several multiplexers, and additional circuitry to implement high-speed adders, subtractors, and shift registers. Two slices form a *configurable logic block* (*CLB*) as shown in Figure 11.39. The CLB forms the basic block used to build the logic circuitry.

* *Antifuse refers to a programmable chip technology that creates permanent, conductive paths between transistors. In contrast to "blowing fuses" in the fusible link method, which opens a circuit by breaking apart a conductive path, the antifuse method closes the circuit by "growing" a conductive path. Two metal layers sandwich a layer of non-conductive, amorphous silicon. When voltage is applied to this middle layer, the amorphous silicon is turned into polysilicon, which is conductive.*

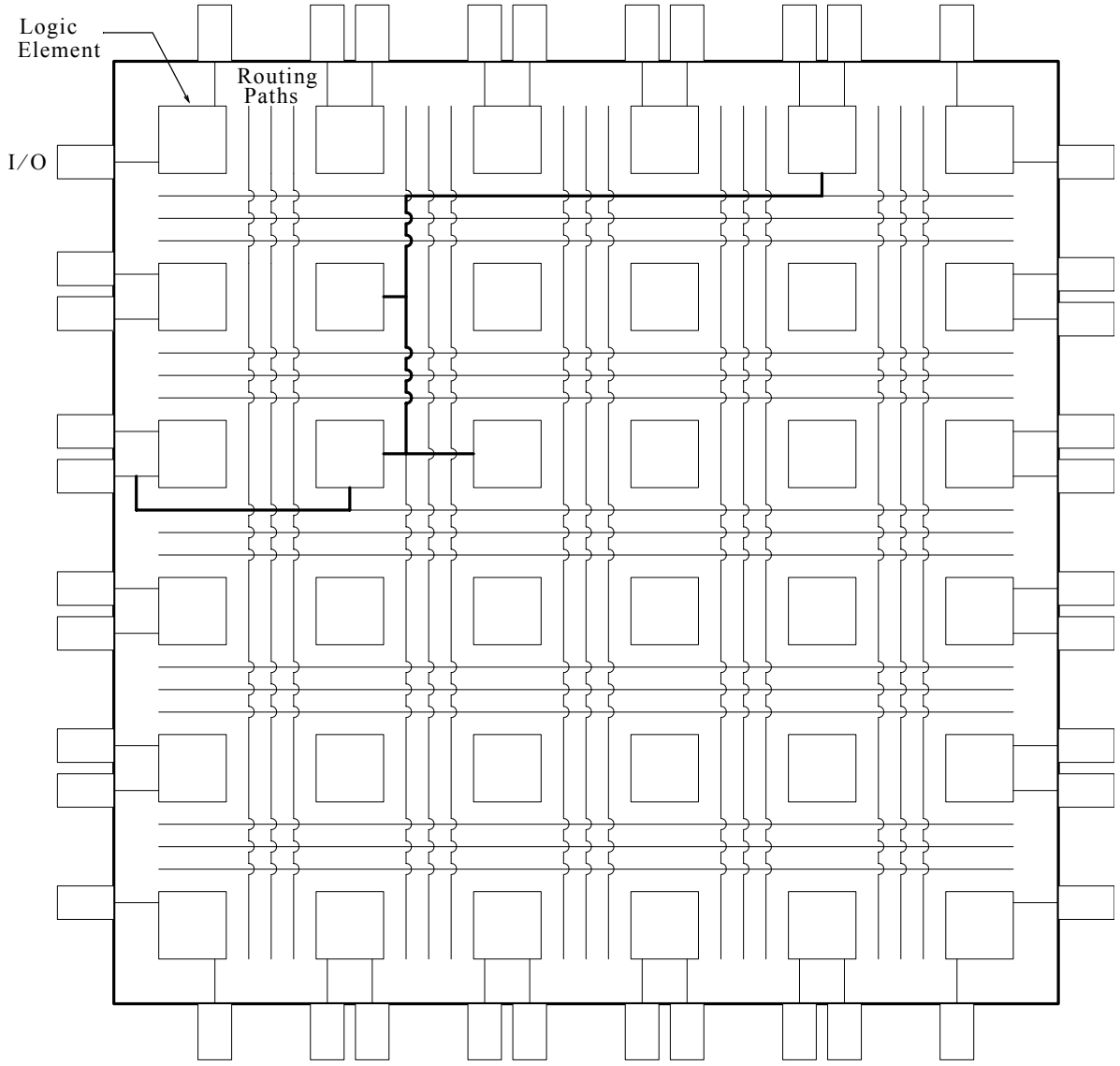


Figure 11.38. Typical SRAM-based FPGA

Some FPGAs, like the Xilinx Virtex families, supply on-chip block RAM. Figure 11.40 shows the Xilinx Virtex CLB matrix that defines a Virtex FPGA. The Virtex family provides a family of devices offering 768 to 19,200 logic slices, and from 8 to 160 variable-form factor block memories.

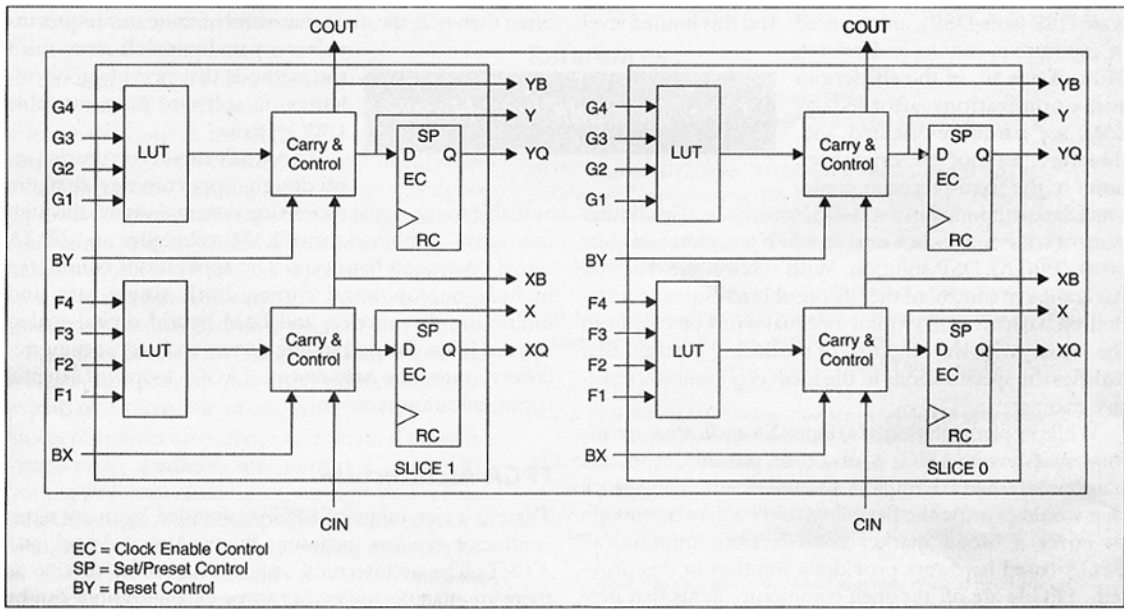


Figure 11.39. Simplified Xilinx Virtex family Configurable Logic Block (CLB) SRAM-based FPGA

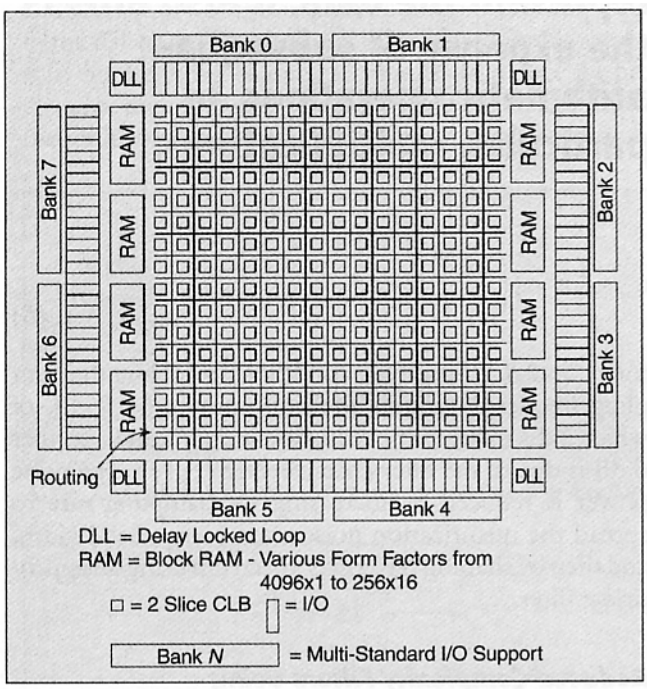


Figure 11.40. Architecture of the Xilinx Virtex family CLB cell array of SRAM-based FPGAs

The Xilinx XC4000 and Virtex devices also allow the designer to use the logic element LUTs as memory – either ROM or RAM. Constructing memory with this distributed approach can yield bandwidths in the order of tens of gigabytes per second range. Typical clock frequencies for current-generation devices are in the 100 to 200 megahertz range.

11.4.1.2 Atmel FPGAs

The Atmel AT40KAL family of FPGAs are also SRAM-based devices and they range in size from 5,000 to 50,000 usable gates. Like the Xilinx Virtex devices, combinational logic is realized using look-up tables. However, the AT40KAL is an 8-sided core with horizontal, vertical, and diagonal cell-to-cell connections implementing fast array multipliers without using busing resources. The logic block architecture employed in the Atmel AT4OKAL FPGA is shown in Figure 11.41.

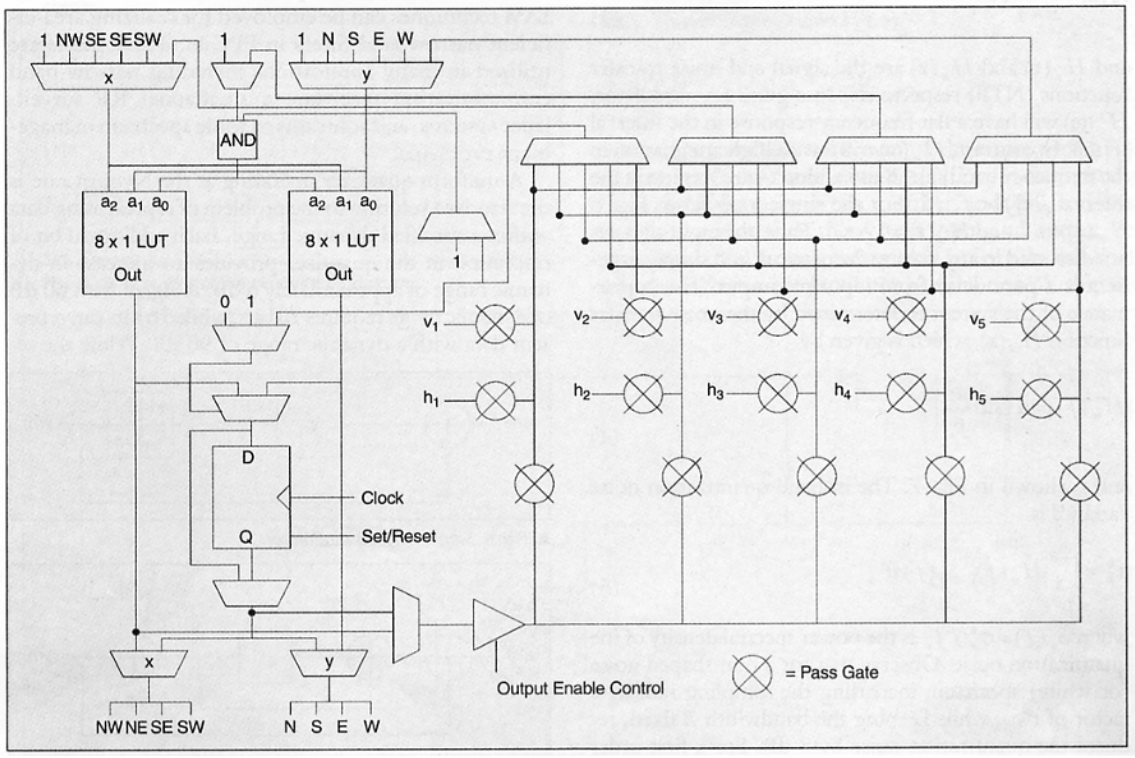


Figure 11.41. Architecture of Atmel's logic cell in the AT40KAL SRAM-based FPGA

As shown in Figure 11.41, two three-input LUTs and a single flip-flop are available in each logic cell. The *pass gates* in a cell form part of the signal routing network, and are used for connecting signals to the multiple horizontal and vertical bus planes. In addition to the orthogonal routing resources, indicated as N, S, E, and W in Figure 11.41, a diagonal group of interconnects (NW, NE, SE, and SW), associated with each cell *x* output, are available to provide efficient connections to a neighboring cell's *x* bus inputs.

11.4.1.3 Altera FPGAs

Altera's Stratix II device family is the newest high-density FPGA family from Altera. Stratix II devices feature a brand new, unique architecture comprised of an innovative logic structure, allowing designers to pack more functionality into less space, thereby reducing developments costs. Combined with the 90-nm process technology, the Stratix II family delivers on average 50

percent faster logic performance, more than twice the logic capacity, and is typically 40 percent lower cost than first generation Stratix devices allowing designers to leverage the advantages of programmable technology in a much wider set of applications. The architecture of the Stratix II FPGA is shown in Figure 11.42.

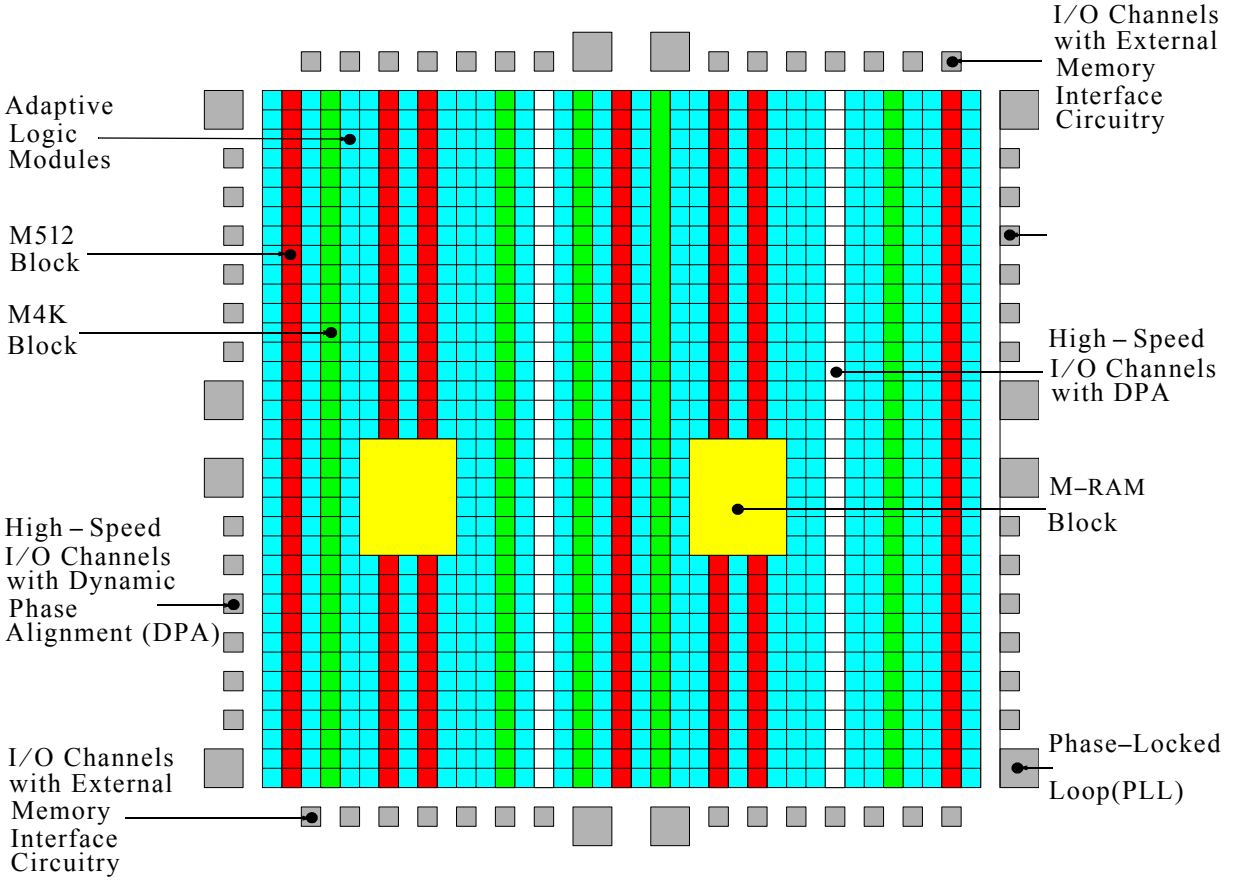

I/O Channels with External Memory Interface Circuitry
Adaptive Logic Modules
M512 Block
M4K Block
High – Speed I/O Channels with DPA
M–RAM Block
High – Speed I/O Channels with Dynamic Phase Alignment (DPA)
Phase–Locked Loop(PLL)
I/O Channels with External Memory Interface Circuitry

Figure 11.42. Architecture of Altera's Stratix II SRAM-based FPGA

The *adaptive logic module* (*ALM*) shown in Figure 11.42 is the fundamental innovation in the Stratix II architecture. An ALM is the basic building block of logic in the Stratix II architecture. It provides advanced features with efficient logic utilization and fast performance. Each ALM contains two *adaptive look-up tables* (*ALUTs*). With up to eight inputs to the combinational logic block, one ALM can implement up to two independent functions each of varying widths. One ALM can also implement any function of up to six inputs and certain seven-input functions. In addition to the two ALUTs, each ALM contains two programmable registers, two dedicated full adders, a carry chain, an adder tree chain, and a register chain that make more efficient use of device logic capacity. Stratix II devices have more than twice the logic of Stratix FPGAs with close to 180,000 equivalent logic elements (LEs). The *dynamic phase alignment* (*DPA*) circuitry accelerates performance by dynamically resolving external board and internal device skew.

Stratix II devices were designed in concert with the Quartus® II software which is considered to

be one of the most advanced development software for high-density FPGAs and provides a comprehensive suite of synthesis, optimization, and verification tools in a single, unified design environment.

11.4.1.4 Lattice FPGAs

The Lattice Semiconductor FPGAs are based on the *Optimized Reconfigurable Cell Array* (*ORCA*) architecture. The basic ORCA block contains an array of *Programmable Function Units (PFUs)*. The structure of an PFU is shown in Figure 11.43.

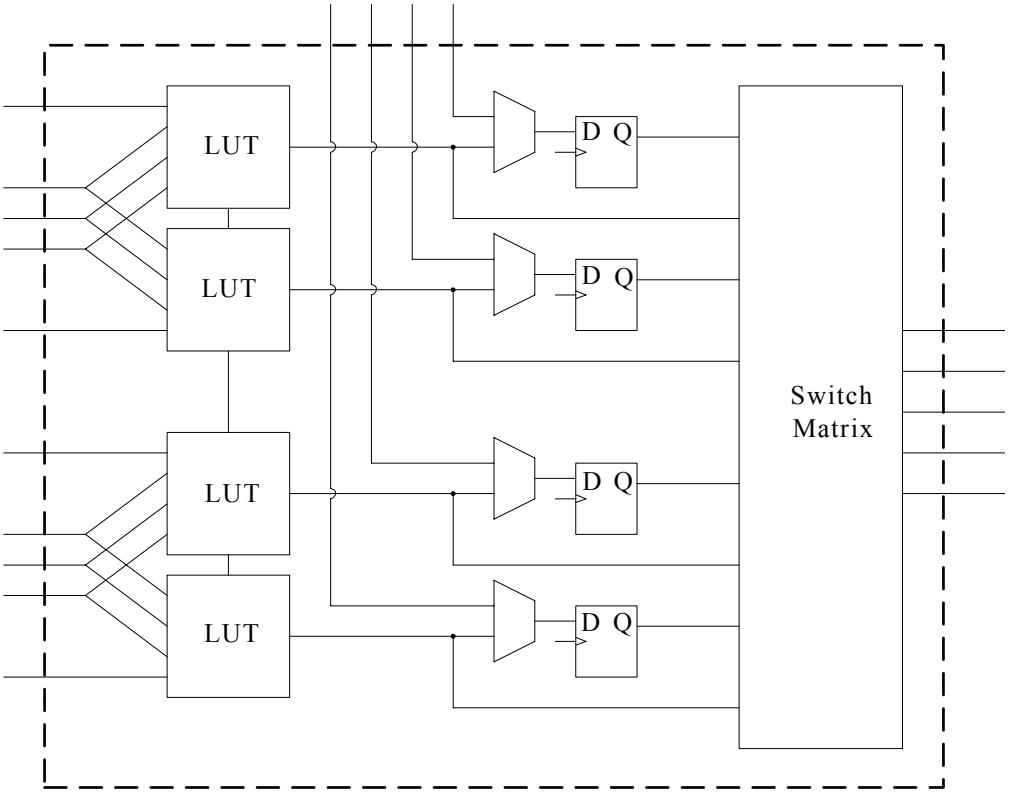


Figure 11.43. Structure of Programmable Function Unit (PFU)

The FPU shown in Figure 11.43 can be configured as four 4-inputs LUTs, or as two 5-input LUTs, or as one 6-input LUT. The latest Lattice Semiconductor ORCA FPGA family offers many features and architectural enhancements not available in earlier FPGA generations. Bringing together highly flexible SRAM-based programmable logic, powerful system features, a rich hierarchy of routing and interconnect resources, and meets multiple interface standards. The new FPGA version, referred to as *ispXPGA*, contains many of the same features of today's mainstream FPGAs and can also be self-configured in less than 200 microseconds by using its electrically-erasable cells which allow nonvolatile reprogrammability. The ispXPGA supports 1,000 electrically-erasable cells.

The programmable function unit (PFU) is based on a four-input lookup table structure and includes dedicated hardware for adders, multipliers, multiplexers, and counters. These programmable units are enmeshed in a segmented routing scheme that includes several types of connections between individual PFUs, long connects across the chip (horizontally and vertically) and feedback signals in each PFU without using external routing. Like most of today's newest FPGAs, the ispXPGA architecture has blocks of embedded memory. These come in the form of strips of block RAM within the PFU array and adjacent to the I/O. The inclusion of embedded memory is also incorporated in Lattice's new line of CPLDs.

11.4.2 Antifuse-Based FPGAs

As stated earlier, the antifuse method closes the circuit by creating a conductive path. Two metal layers sandwich a layer of non-conductive, amorphous silicon. When voltage is applied to this middle layer, the amorphous silicon is turned into polysilicon, which is conductive.

11.4.2.1 Actel FPGAs

Actel provides the *Axcelerator* family of FPGAs. This family uses a patented metal-to-metal antifuse programmable interconnect that resides between the upper two layers shown in Figure 11.44.

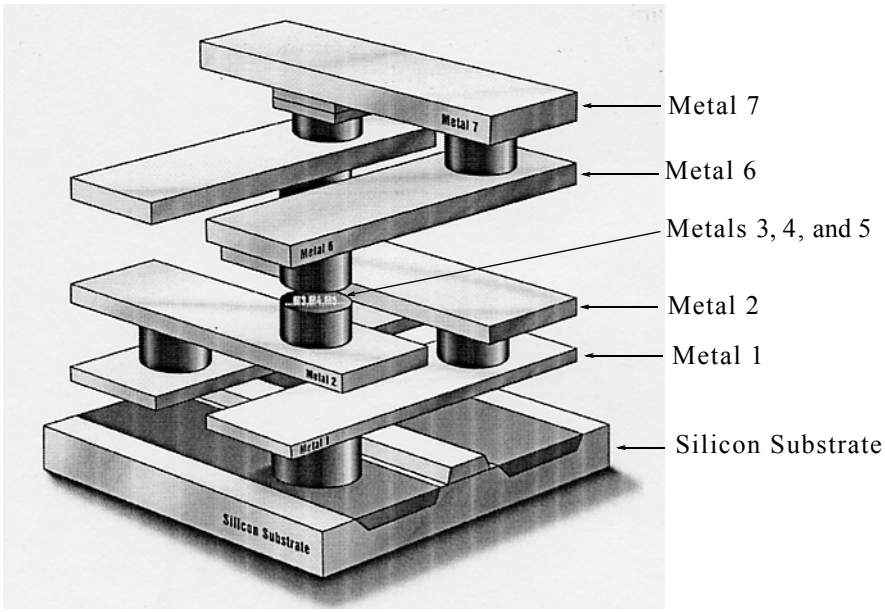


Figure 11.44. Actel's Axcelerator Family Interconnect Elements

The architecture of Actel's Axcelerator family of FPGAs is based on the so-called *sea-of-modules architecture.* A comparison of the traditional FPGA architecture and the sea-of-modules architecture is shown in Figure 11.45.

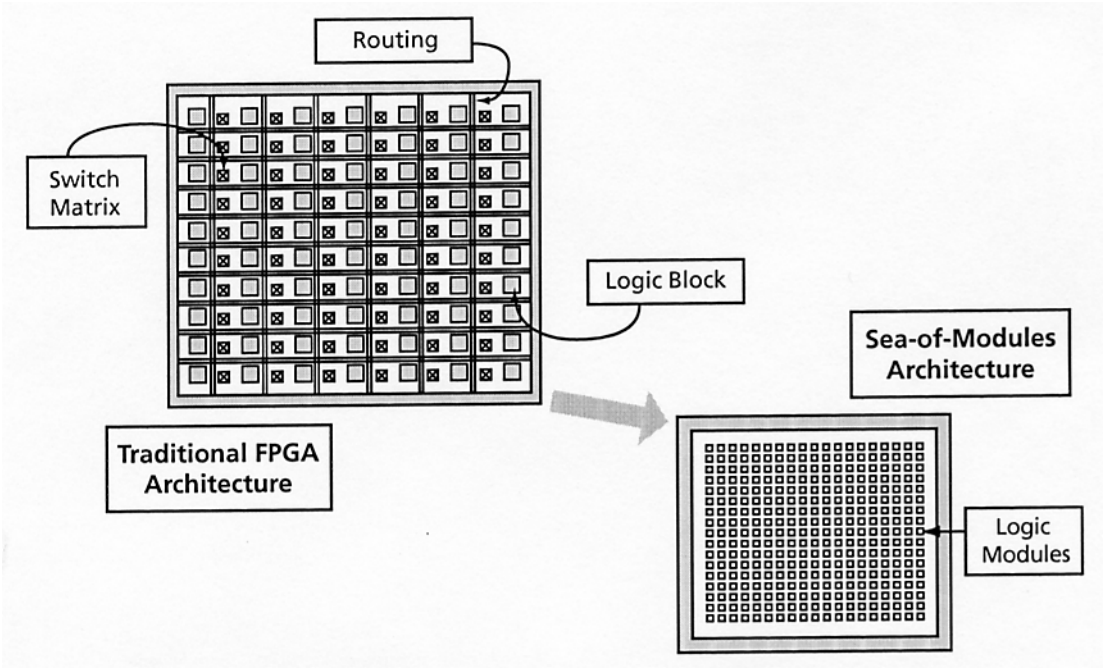


Figure 11.45. A comparison of the traditional FPGA architecture and the sea-of-modules architecture

Actel's Axcelerator family of FPGAs provides two types of logic modules: the combinational cell (C-cell), and the register cell (R-cell) shown in Figure 11.46.

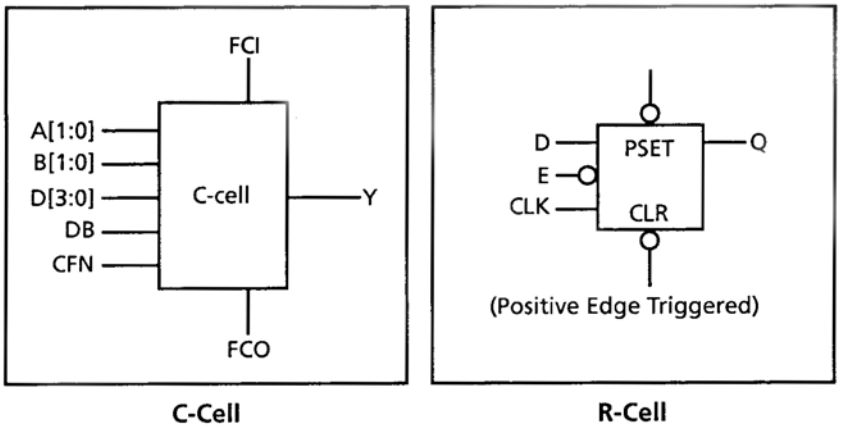


Figure 11.46. Logic modules in Actel's Axcelerator family of FPGAs

The C-cell can implement more than 4,000 combinational functions of up to five inputs, and the R-cell contains a flip-flop featuring asynchronous clear, asynchronous preset, and active-low enable control signals The R-cell registers feature programmable clock polarity selectable on a register-by-register basis. This provides additional flexibility (e.g., easy mapping of dual-data-rate functions into the FPGA) while conserving valuable clock resources. The clock source for the R-

cell can be chosen from the hardwired clocks, routed clocks, or internal logic.

Two C-cells, a single R-cell, two Transmit (TX), and two Receive (RX) routing buffers form a *cluster*, while two clusters comprise a *supercluster* as shown in Figure 11.47.

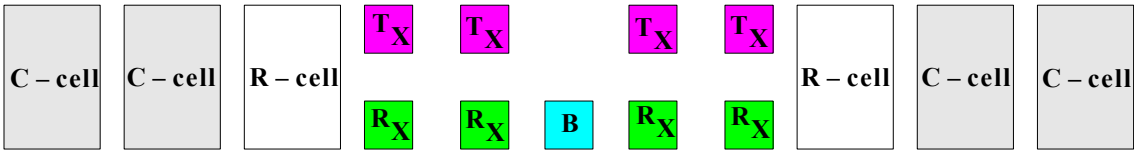


Figure 11.47. Actel's supercluster

Each supercluster also contains an independent buffer (B) module, which supports buffer insertion on high fan-out* circuits by the place-and-route tool, minimizing system delays while improving logic utilization. The logic modules within the supercluster are arranged so that two combinational modules are side-by-side, giving a C–C–R – C–C–R pattern to the supercluster. This C–C–R pattern enables minimum delay of two-bit carry logic for improved arithmetic performance as shown in Figure 11.48.

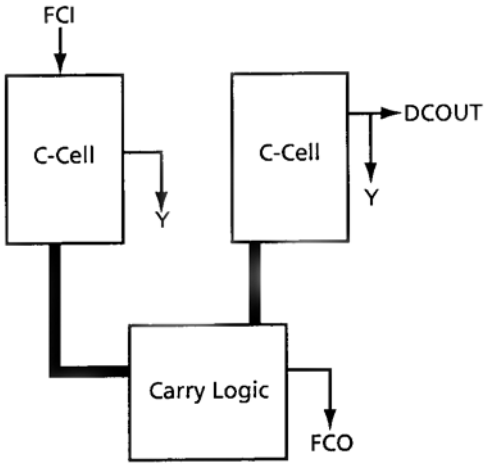


Figure 11.48. Actel's 2-bit carry logic

The Axcelerator architecture is fully fracturable, meaning that if one or more of the logic modules in a supercluster are used by a particular signal path, the other logic modules are still available for use by other paths. At the chip level, superclusters are organized into core tiles, which are arrayed to build up the full chip. For example, the AX1000 is composed of a 3x3 array of nine core tiles. The number of core tiles in the Axcelerator family of FPGAs are shown in Table 11.3.

* *Fan-out is a term that defines the maximum number of digital inputs that the output of a single logic gate can feed. Generally, TTL gates can feed up to 10 other digital gates or devices. Thus, a typical TTL gate has a fan-out of 10. For more information, refer to Electronic Devices and Amplifier Circuits, ISBN 0-9744239-4-7.*

TABLE 11.3 Number of core tiles per device

Device	Number of Core Tiles
AX125	1 regular tile
AX250	4 smaller tiles
AX500	4 regular tiles
AX1000	9 regular tiles
AX2000	16 regular tiles

Surrounding the array of core tiles are blocks of I/O clusters and the I/O bank ring. Each core tile consists of an array of 336 superclusters and four SRAM blocks (176 superclusters and three SRAM blocks for the AX250). The SRAM blocks are arranged in a column on the west side of the tile as shown in Figure 11.49.

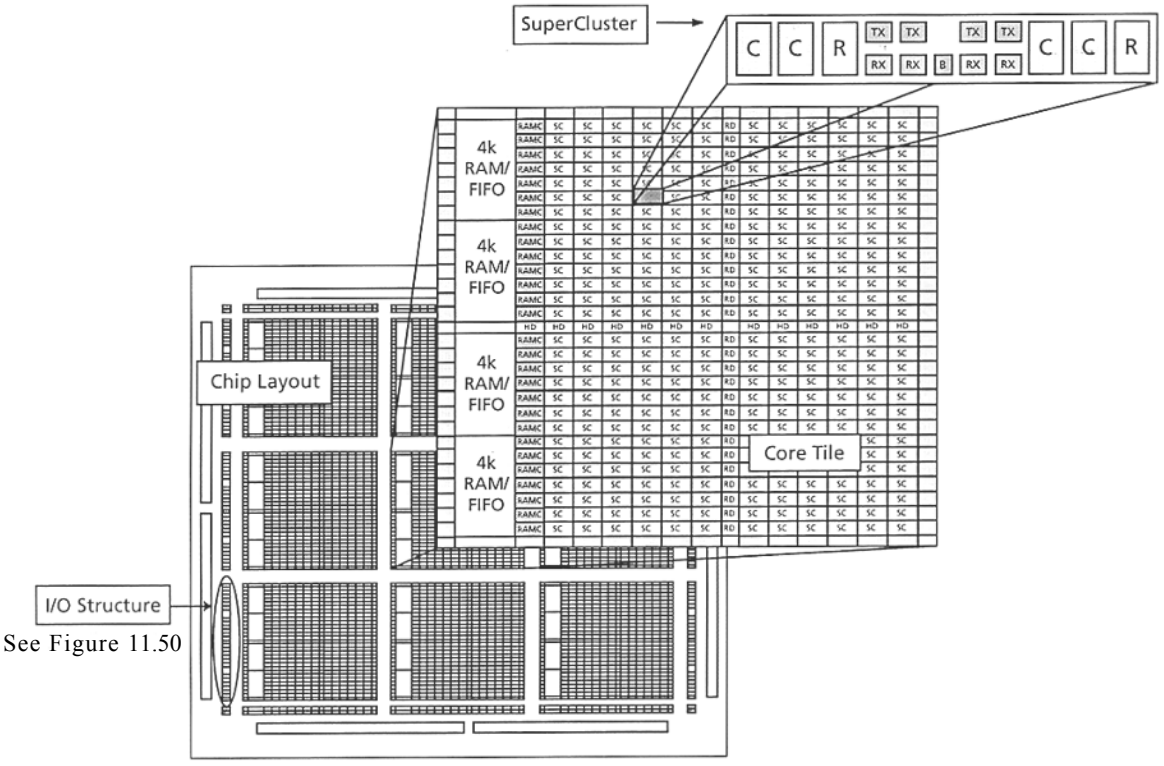


Figure 11.49. Actel's AX1000 device architecture

As mentioned earlier, each core tile has either three (in a smaller tile) or four (in the regular tile) embedded SRAM blocks along the west side, and each variable-aspect-ratio SRAM block is 4,608 bits in size. Available memory configurations are: 128 × 36, 256 × 18, 512 × 9, 1024 × 4, 2048 × 2 or 4096 × 1 bits. The individual blocks have separate read and write ports that can be configured with different bit widths on each port. Data can be written in by eight and read out by one.

In addition, every SRAM block has an embedded first-in, first-out (FIFO) control unit. The FIFO control unit allows the SRAM block to be configured as a synchronous FIFO without using core logic modules. The FIFO width and depth are programmable. The FIFO also features programmable ALMOST-EMPTY (AEMPTY) and ALMOST-FULL (AFULL) flags in addition to the normal EMPTY and FULL flags. In addition to the flag logic, the embedded FIFO control unit also contains the counters necessary for the generation of the read and write address pointers as well as control circuitry to prevent metastability and erroneous operation. The embedded SRAM/FIFO blocks can be cascaded to create larger configurations.

The Axcelerator family of FPGAs features a flexible I/O structure, supporting a range of mixed voltages with its bank-selectable I/Os: 1.5V, 1.8V, 2.5V, and 3.3V. Axcelerator FPGAs support at least 14 different I/O standards (single-ended, differential, voltage-referenced). The I/Os are organized into banks, with eight banks per device (two per side). The configuration of these banks determines the I/O standards supported. All I/O standards are available in each bank. Each I/O module has an input register (InReg), an output register (OutReg), and an enable register (EnReg) as shown in Figure 11.50.

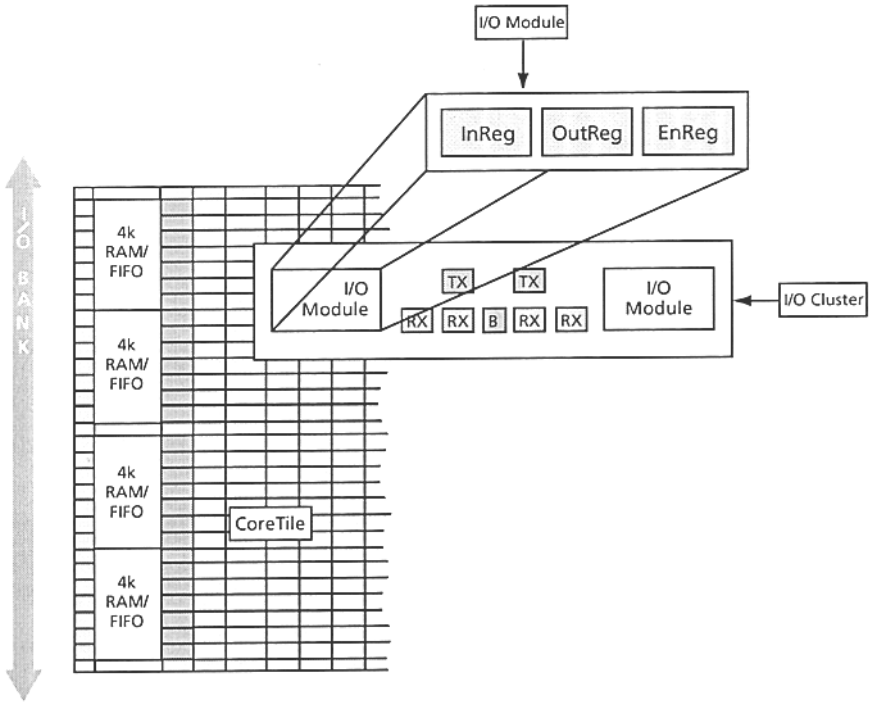


Figure 11.50. Actel's Axcelerator I/O cluster arrangement

An I/O cluster includes two I/O modules, four RX modules, two TX modules, and a buffer (B) module.

The hierarchical routing structure for Actel's Axcelerator family of FPGAs is shown in Figure 11.51.

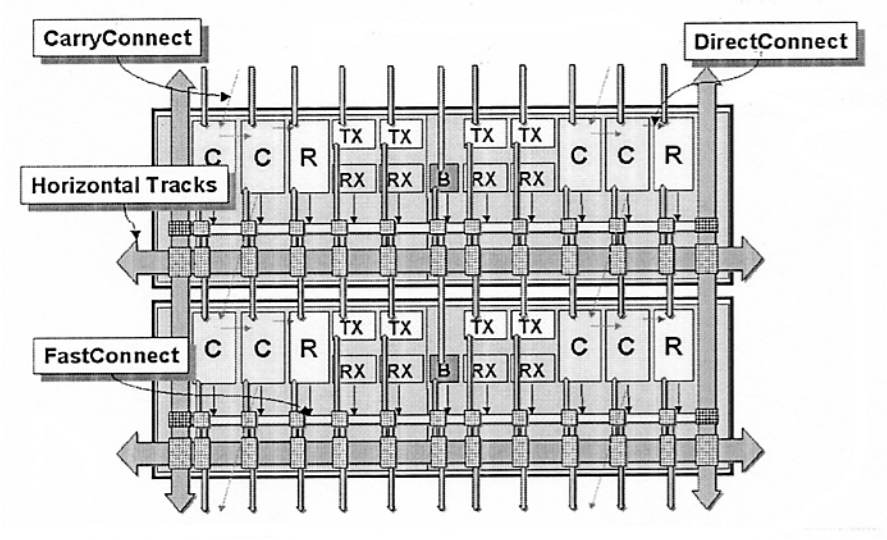


Figure 11.51. Actel's Axcelerator Routing Structure

As shown in Figure 11.51, the routing structure connects the logic modules, the embedded memory blocks, and the I/O modules in one block. At the lowest level, in and between superclusters, there are three local routing structures: FastConnect, DirectConnect, and CarryConnect routing. DirectConnects provide the highest performance routing inside the superclusters by connecting a C-cell to the adjacent R-cell. DirectConnects do not require an antifuse to make the connection and achieve a signal propagation time of less than 0.1 ns.

FastConnects provide high-performance, horizontal routing inside the supercluster and vertical routing to the supercluster immediately below it. Only one programmable connection is used in a FastConnnect path, delivering a maximum routing delay of 0.4 ns. CarryConnects are used for routing carry logic between adjacent superclusters. They connect the FCO (see Figure 11.48) output of one two-bit, C-cell carry logic to the FCI input of the two-bit, C-cell carry logic of the supercluster below it. CarryConnects do not require an antifuse to make the connection and achieve a signal propagation time of less than 0.1 ns.

The next level contains the core tile routing. Over the superclusters within a core tile, both vertical and horizontal tracks run across rows or columns, respectively. At the chip level, vertical and horizontal tracks extend across the full length of the device, both north-to-south and east-to-west. These tracks are composed of highway routing that extend the entire length of the device (segmented at core tile boundaries) as well as segmented routing of varying lengths.

Each family member has three types of global signals available to the designer: HCLK, CLK, and GCLR/GPSET. There are four hardwired clocks (HCLK) per device that can directly drive the clock input of each R-cell. Each of the four routed clocks (CLK) can drive the clock, clear, preset, or enable pin of an R-cell or any input of a C-cell as shown in Figure 11.46.

Global clear (GCLR) and global preset (GPSET) drive the clear and preset inputs of each R-cell

as well as each I/O Register on a chip-wide basis at power-up. Each HCLK and CLK has an associated analog PLL (a total of eight per chip). Each embedded PLL can be used for clock delay minimization, clock delay adjustment, or clock frequency synthesis. The phase-locked loop (PLL) is capable of operating with input frequencies ranging from 14 MHz to 200 MHz and can generate output frequencies between 20 MHz and 1 GHz. The clock can be either divided or multiplied by factors ranging from 1 to 64. Additionally, multiply and divide settings can be used in any combination as long as the resulting clock frequency is between 20 MHz and 1 GHz. Adjacent PLLs can be cascaded to create complex frequency combinations.

The PLL can be used to introduce either a positive or a negative clock delay of up to 3.75 ns in 250 ps increments. The reference clock required to drive the PLL can be derived from three sources: external input pad (either single-ended or differential), internal logic, or the output of an adjacent PLL.

Actel's Axcelerator family of FPGAs was created for high-performance designs but also includes a low power mode (activated via the LP pin). When the low power mode is activated, I/O banks can be disabled (inputs disabled, outputs tristated), and PLLs can be placed in a power-down mode. All internal register states are maintained in this mode. Furthermore, individual I/O banks can be configured to opt out of the LP mode, thereby giving the designer access to critical signals while the rest of the chip is in low power mode.

The Axcelerator family of FPGAs is fully supported by both Actel's Libero™ Integrated Design Environment (IDE) and Designer FPGA Development software. Actel Libero IDE is an integrated design manager that seamlessly integrates design tools while guiding the user through the design flow, managing all design and log files, and passing necessary design data among tools. Additionally, Libero IDE allows users to integrate both schematic and HDL synthesis into a single flow and verify the entire design in a single environment. Libero IDE includes Synplify® Actel Edition (AE) from Synplicity® , ViewDraw® AE from Mentor Graphics® , ModelSim® HDL Simulator from Mentor Graphics, WaveFormer Lite™ AE from SynaptiCAD® , and Designer software from Actel.

11.4.2.2 QuickLogic FPGAs

QuickLogic* introduced three families of FPGAs – pASIC-1, pASIC-2, and pASIC-3® built on the patented ViaLink® metal-to-metal interconnect antifuse technology. QuickLogic announced recently that it is permanently discontinuing the manufacture of the pASIC-1 and pASIC-2 families. Accordingly, our discussion will be on pASIC-3.

QuickLogic's pASIC-3 family of FPGAs range from 4,000 to 60,000 usable PLD gates in 75 to 316 I/Os. The pASIC-3 devices are manufactured on a 0.35 micron 4-layer metal process using QuickLogic's patented ViaLink. Members of the pASIC-3 family feature 3.3 volt operation with 5 volt compatibility, and are available in commercial, industrial, and military temperature grades. The architecture is similar to that of the pASIC-2† family, though pASIC-3 devices are faster and

* *QuickLogic should not be confused with another company, Qlogic.*

lower cost as a result of their smaller process geometry. The pASIC-3 family also provides high system performance – with 16-bit counter speeds of up to 300 MHz and data path speeds over 400 MHz.

As shown in Figure 11.52, the pASIC-3 family of devices contain a range of 96 to 1,584 logic cells.

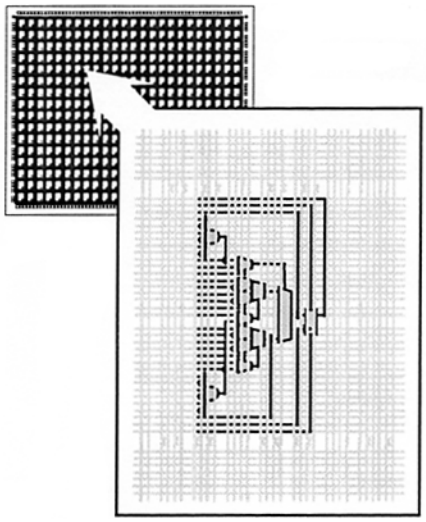


Figure 11.52. QuickLogic's pASIC-3 logic cell arrangement

Software support for the complete pASIC-3 family is available through two basic packages. The turnkey Quick Works® package provides the most complete FPCA software solution from design entry to logic synthesis, to place and route, to simulation. The QuickTools™ for Workstations package provides a solution for designers who use Cadence® , Exemplar™, Mentor® , Synopsys® , Synplicity® , Viewlogic™, Aldec™, or other third-party tools for design entry, synthesis, or simulation.

11.5 FPGA Block Configuration - Xilinx FPGA Resources

Before programming an FPGA, it is necessary to view and define a block's configurable parameters. These parameters include:

1. Arithmetic type – unsigned or twos complements
2. Latency – specify the delay through the block
3. Overflow and quantization – the user can saturate or wrap overflow, and can select truncate quantization or rounding quantization.
4. Precision – full precision or the user can define the number of bits and where the binary point

† *pASIC-2 was second-sourced by Cypress Semiconductor.*

is for the block.

5. Sample period – an integer value

6. Equations – to simplify and speed up calculations

While most programs, MATLAB, Simulink, C++, and others use double-precision arithmetic which we discussed in Chapter 3, some FPGA manufacturers such as Xilinx use *n*-bit fixed point numbers specified as

$$\text{Format} = \text{Sign\_Width\_Binary point from the LSB}$$

where

$$\text{Sign} = \begin{cases} \text{Fix} = \text{Signed Value} \\ \text{UFix} = \text{Unsigned Value} \end{cases}$$

Example 11.4

Specify the format and give the signed numerical value of the binary word in Figure 11.53.

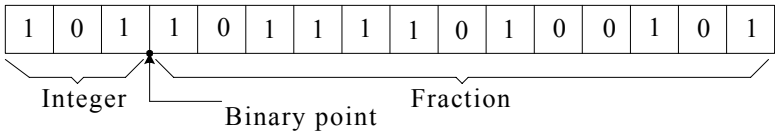


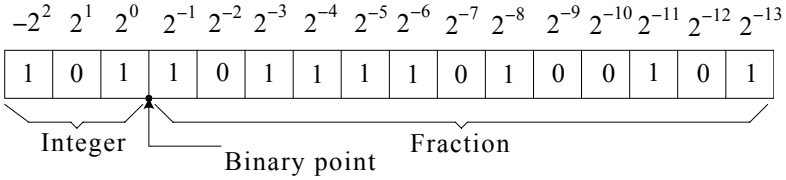
Figure 11.53. Signed binary word for Example 11.4

Solution:

This is a signed number, whose width (integer part + fractional part = 3 + 13 = 16) is 16 bits, and the decimal point from the LSB is 13 bits. Therefore,

$$\text{Format} = \text{Fix\_16\_13}$$

The weights for this signed number are shown below.



The decimal equivalent of the binary weights is

$$\begin{aligned} &-2^2 + 2^0 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-6} + 2^{-8} + 2^{-11} + 2^{-13} \\ &= -4 + 1 + \frac{1}{2} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{256} + \frac{1}{2048} + \frac{1}{8192} \\ &= -2.261108\ldots \end{aligned}$$

A conversion is required whenever there is a communication between Simulink blocks and Xilinx blocks. Xilinx FPGA programming includes blocks referred to as *Gateway In and Out blocks*. Their symbols are shown in Figure 11.54.

Figure 11.54. Symbols for conversion from double to N-bit fixed point and vice versa

A conversion example is shown in Figure 11.55.

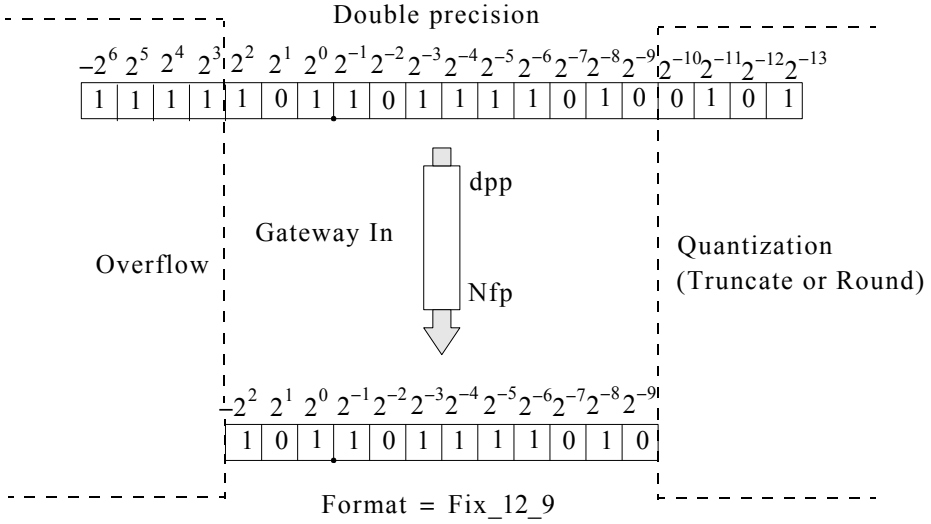


Figure 11.55. Example of conversion from double precision to N-bit fixed point precision

Quantization occurs if the number of fractional bits is insufficient to represent the fractional portion of a value. We can choose either to *truncate* by discarding the bits to the right of the least significant bit, or to *round* to the nearest representable value or to the value farthest from zero if there are two equidistant nearest representable values.

Example 11.5

The double-precision value of a number is shown in Figure 11.56.

Figure 11.56. Double-precision value for the number of Example 11.5

a. Truncate this number to the format <Fix_12_9> and give its decimal equivalent.

b. Round this number to the format <Fix_12_9> and give its decimal equivalent.

Solution:

The truncated and rounded outputs of this number are shown in Figure 11.57.

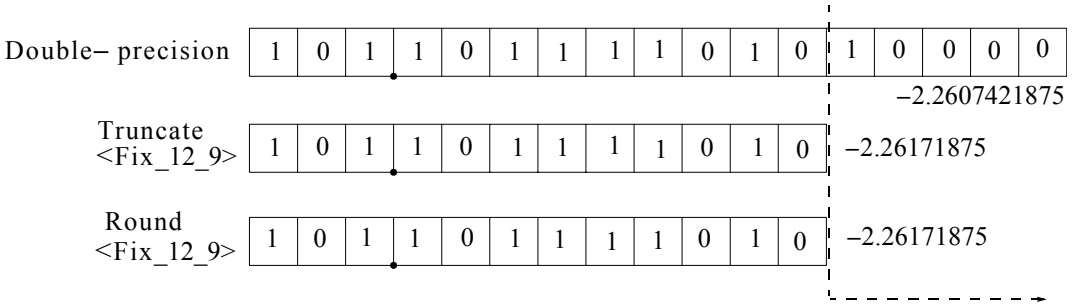


Figure 11.57. Truncated and rounded values for the double precision number of Example 11.5

A positive double-precision number will have different output depending on whether truncation or rounding is employed.

Example 11.6

The positive double-precision value of a number is shown in Figure 11.58.

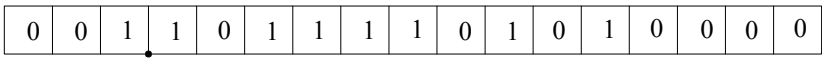


Figure 11.58. Positive double-precision value for the number of Example 11.6

a. Truncate this number to the format <Fix_12_9> and give its decimal equivalent.

b. Round this number to the format <Fix_12_9> and give its decimal equivalent.

Solution:

The truncated and rounded outputs of this number are shown in Figure 11.59.

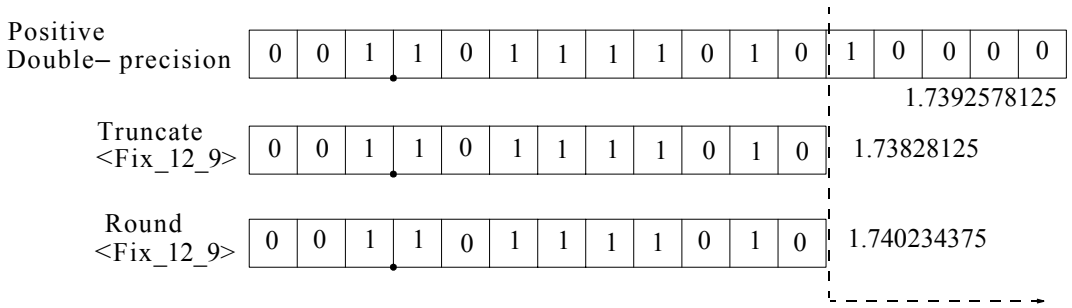


Figure 11.59. Truncated and rounded values for the positive double precision number of Example 11.6

An unsigned double-precision number will have different output depending on whether truncation or rounding is employed.

Example 11.7

The unsigned double-precision value of a number is shown in Figure 11.60.

Figure 11.60. Unsigned double-precision value for the number of Example 11.7

Solution:

The truncated and rounded outputs of this number are shown in Figure 11.61.

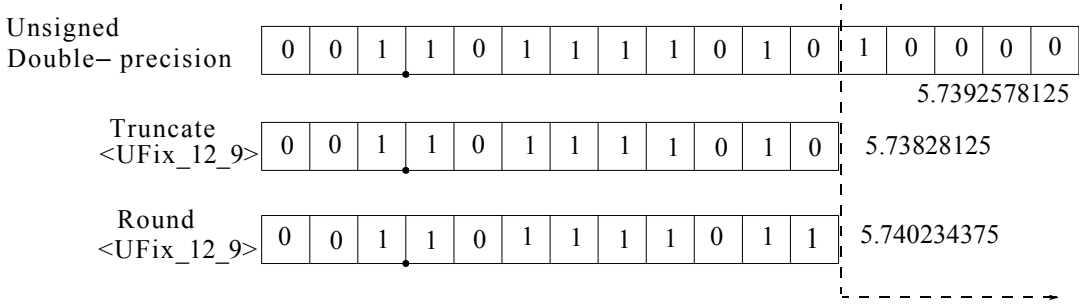


Figure 11.61. Truncated and rounded values for the unsigned double precision number of Example 11.7

An *overflow* occurs when a value lies outside the specified range. The user can choose to *flag* an overflow, or to *saturate* to the largest positive (or maximum negative value), or to *wrap* the value, that is, discard any significant bits beyond the most significant bit in the fixed point number.

Example 11.8

The positive double-precision value of a number is shown in Figure 11.62.

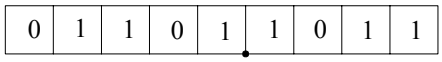


Figure 11.62. Double-precision value for the number of Example 11.8

a. Saturate this number to the format <Fix_7_4> and give its decimal equivalent.

b. Wrap this number to the format <Fix_7_4> and give its decimal equivalent.

Solution:

The saturated and wrapped outputs of this number are shown in Figure 11.63.

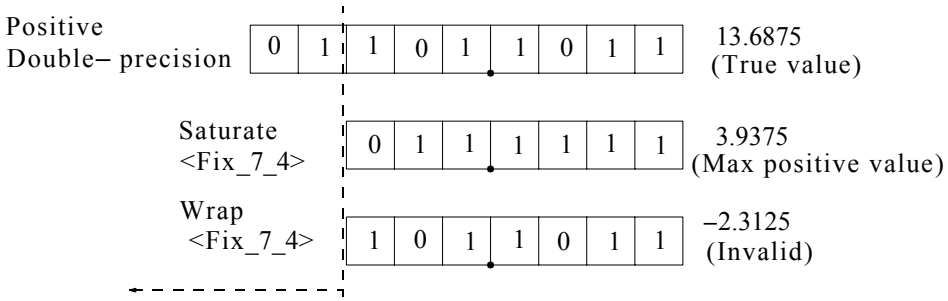


Figure 11.63. Saturated and wrapped values for the double precision number of Example 11.8

The wrapping operation in Example 11.8 produced the negative value –2.3125 in *n*-bit fixed point number format, and obviously this is an invalid number. This problem is alleviated with the use of blocks that have a *valid bit input* that is a control signal to the data input. The blocks are also provided with a valid bit output that produces a NaN (not a number) output if the output data are invalid. In most cases the bits are valid since data is processed with Finite Input Response* (FIR), Fast Fourier Transform (FFT), Reed-Salomon encoder / decoder, convolutional encoder, and interleaver/deinterleaver blocks.

FPGAs contain several blocks for concatenation of data, conversion, reinterpret, and other operations. We will discuss four important blocks contained in Xilinx's FPGAs. These are shown in Figure 11.64.

* *It is beyond the scope of this text to discuss these advanced digital signal processing and information theory topics. For detailed discussion on FIR and FFT, please refer to Signals and Systems with MATLAB Applications, ISBN 0-9709511-6-7.*

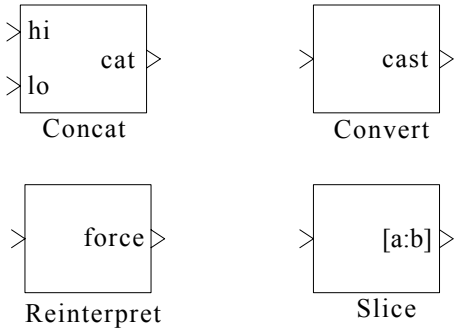


Figure 11.64. Xilinx's block symbols

The *Concat block* performs the concatenation of two binary vectors. Both vectors must be unsigned integers, that is, unsigned numbers with binary points at position zero. The Reinterpret block, discussed below, provides capabilities that can extend the functionality of the Concat block.

The *Convert block* converts each input to a number of a desired arithmetic type. For instance, a number can be converted to a signed (twos complement) or unsigned value. The total number of bits and binary point are specified by the user. Also, quantization (truncate and rounding) can be applied to the output value. First, it lines up the binary point between input and output. Next, the total number of bits and binary point the user specifier are used, and if overflow and quantization options are chosen, the output may change.

Example 11.9

Give an example to show how a Convert block could be used to represent the same value using different number of bits and binary point.

Solution:

A possible arrangement is shown in Figure 11.65.

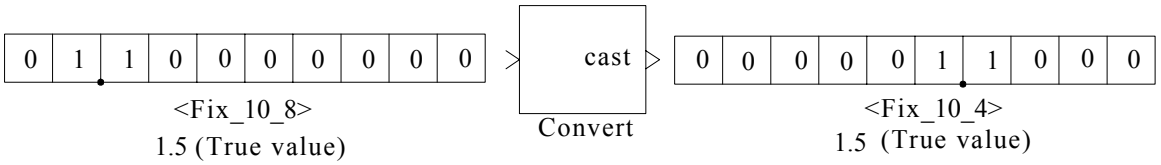


Figure 11.65. Convert operation with Xilinx's Convert Block for Example 11.9

Saturating the overflow in a Convert block, may cause a change in the fractional part of a number. Truncation and rounding may also affect the value to the left of the binary number.

Example 11.10

The true value of 1.5 is in <Fix_10_8> format. A Convert block is used to convert this number to <Fix_6_0> format. Determine the truncated and rounded values at the output of the Convert block.

Solution:

The input in <Fix_10_8> format and the truncated and rounded outputs in <Fix_6_0> format are shown in Figure 11.66.

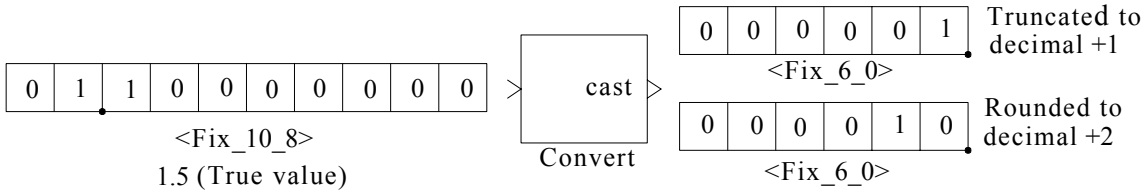


Figure 11.66. Convert operation with Xilinx's Convert Block for Example 11.10

The *Reinterpret block* forces the output to a new type without any regard for retaining the numerical value applied at the input. However, the total number of bits at the output is the same as the total number of inputs. This block allows unsigned data to be reinterpreted as signed data or vice versa. It also allows scaling of the data through the repositioning of the binary point.

Example 11.11

The true value of 1.5 is in <Fix_10_8> format. A Reinterpret block is used to force the binary point from position 2 to position 5. Determine the output format and its numerical value.

Solution:

The input in <Fix_10_8> format and the truncated and rounded outputs in <Fix_6_0> format are shown in Figure 11.67.

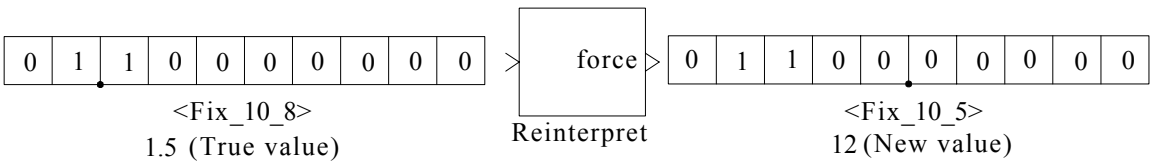


Figure 11.67. Reinterpret operation with Xilinx's Reinterpret Block for Example 11.11

The *Slice block* allows the user to slice off a sequence of bits from the input data and create a new data value. The output data is unsigned with the binary point at the zero position. Figure 11.68 shows the result after taking a 4-bit slice from a number with the <Fix_10_8> format.

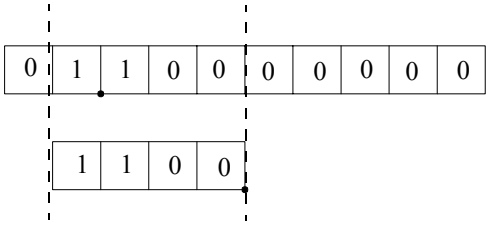


Figure 11.68. Slice operation with Xilinx's Reinterpret Block for Example 11.11

In Figure 11.68 the slice operation calls for taking a 4-bit slice from the <Fix_10_8> number and offsetting the bottom (LSB) bit of the slice by 5 bits.

11.6 The CPLD versus FPGA Trade-Off

CPLDs are typically faster and have more predictable timing than FPGAs. However, FPGAs are generally more dense and contain more flip-flops and registers than CPLDs. Operating speed generally decreases as the PLD density increases, however. Our design may fit into a CPLD; if not, we must use an FPGA. With CPLDs, we should try to fit as much logic as possible into a macrocell to take advantage of the CPLD's predictability. When we design with an FPGA, on the other hand, the devices' large fan-outs from one logic cell to another can cause long delays. Placing two buffers at the output of the logic cell lets us split the fan-out to decrease the overall delay even after adding the buffer to the output.

Applications of CPLDs and FPGAs include, but not limited to, coprocessors for high speed designs, finite impulse response (FIR) filters, fast Fourier transforms (FFT), and discrete cosine transforms (DCT) that are required for video compression and decompression, encryption, convolution, and forward error correction (FEC).* The most commonly used high level languages are Verilog and VHDL. Most manufacturers provide free integrated development environments (IDEs) and tools.

11.7 What is Next

The material in this text reflects the latest in state-of-the art technology. Undoubtedly, newer fastest, and denser products will replace the present devices. How does one keep up with the latest technology? The answer is through the Internet, preferably getting on manufacturers lists for obtaining information on new releases via e-mail. Also, some manufacturers promote their products by offering free in-house seminars. Free information on Xilinx products can be obtained through the Xilinx University Program and Workshops. In some cases, Xilinx will allow one to download the software necessary for completing some of the labs at home. An example in design-

* *FEC is a technique for minimizing transmission errors by adding overhead data to help the decoder interpret the received message correctly. Reed-Salomon is an example of FEC technique.*

ing a *multiply-accumulate* (MAC) circuit is presented below.

As shown in Figure 11.69, a MAC can be implemented as a single engine or multiple engines.

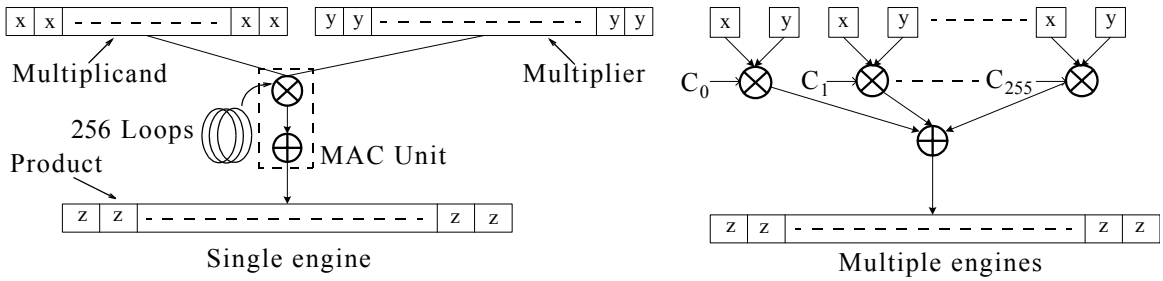


Figure 11.69. Single- and multiple-engine MAC implementation

As shown in Figure 11.69, in the single engine the MAC unit is time-shared, and using a 256-tap FIR filter we need to perform 256 multiply and accumulate operations per data sample, and we get one output every 256 clock cycles. However, with the multiple engines configuration using again a 256-tap FIR filter we can perform 256 multiply and accumulate operations per data sample, but we get one output every clock cycle.

One of the downloadable Xilinx files available at university.xilinx.com pertains to the creation of a 12-bit × 8-bit MAC using VHDL. A block diagram of this 12-bit × 8-bit MAC is shown in Figure 11.70.

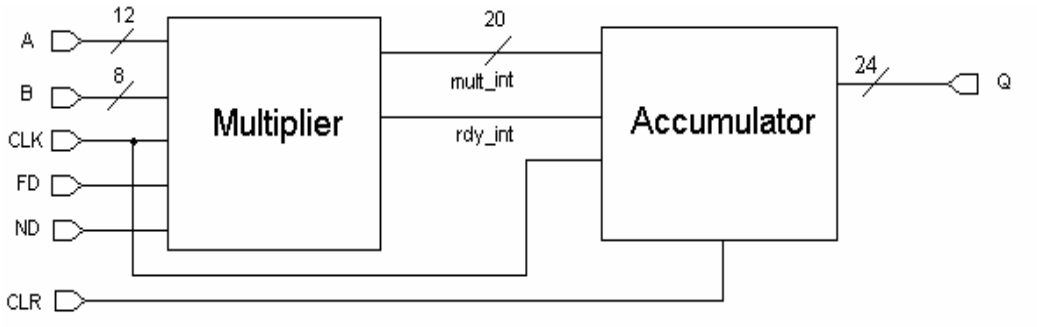


Figure 11.70. 12-bit × 8-bit MAC

The multiply-accumulate operations are as follows:

$$1 \times 1 = 1$$
$$(1 \times 1) + (2 \times 2) = 5$$
$$(1 \times 1) + (2 \times 2) + (3 \times 3) = 14$$
and so on

The VHDL folder contains the file *accum* whose content is as follows:

```
Component accum
  port (
  B: IN std_logic_VECTOR(15 downto 0);
  Q: OUT std_logic_VECTOR(15 downto 0);
  CLK: IN std_logic;
  CE: IN std_logic;
  ACRL: IN std_logic;);
end component;
```

The Xilinx implementation tools include two files known as *Place and Route Report* file, and *Post-Place and Route Timing Report* file. The first displays the number of slices*, number of global buffers/multiplexers (BUFGMUX), and number of external I/O buffers (IOB). Upon completion of this lab, the Place and Route Report file shows that 63 slices, 1 BUFGMUX, and 47 IOBs were used. The Post Place and Route Timing Report file shows that the maximum clock frequency is approximately 90 MHz.

Most of the Xilinx downloaded labs mention the "System Generator for DSP". This is a software tool that Xilinx developed to make Simulink† a powerful hardware design environment.

* *Each Xilinx Virtex-II Slice contains 4 look-up tables (LUTs), 16-bit distributed Select RAM, and a 16-bit shift register.*

† *It is strongly recommended that the reader becomes familiar with MATLAB and Simulink before attempting most of the downloadable Xilinx labs. Both of these software packages are available as Student Versions from the MathWorks, Inc. www.mathworks.com.*

11.8 Summary

- A Programmable Logic Array (PLA) is a small Field Programmable Device (FPD) that contains two levels of logic, AND and OR, commonly referred to as AND-plane and OR-plane respectively. A PLA is similar to a ROM but does not generate all the minterms as a ROM does. The size of a PLA is defined by the number of inputs, the number of product terms in the AND-plane, and the number of the sum terms in the OR-plane.
- PLAs may be mask-programmable or field programmable. With a mask programmable PLA the buyer submits a PLA program table to the manufacturer. With a field programmable PLA, the buyer configures the PLA himself. Programming the PLA implies specifying the paths in its AND-OR-Invert form. When designing digital systems with PLAs, there is no need to show the internal connections; all is needed is a PLA program table.
- The first PLAs, introduced in the 1970s, contained 48 product terms (AND terms) and 8 sum terms (OR terms). The manufacturing costs of these PLAs were high, and because of the two levels of logic (AND and OR), the speed was poor.
- The Programmable Array Logic (PAL) devices were developed to overcome the deficiencies of PLAs. A major difference between PLAs and PALs is that the latter is designed with a programmable wired-AND plane followed by a fixed number of OR gates and flip flops so that sequential circuits could also be implemented.
- Variants of the original PALs were designed with different inputs, and various sizes of OR-gates, and they form the basis for present-day digital hardware designs. Figure 11.6 shows the block diagram of a typical PAL.
- PLAs, PALs, and PAL variants are generally referred to as Simple Programmable Logic Devices (SPLDs).
- Complex Programmable Logic Devices (CPLDs) integrate multiple SPLDs into a single chip. A typical CPLD provides a logic capacity equivalent to about 50 single SPLDs. Most are provided with in-system programmability (ISP) circuitry compatible with IEEE Std 1532.
- In-system programming can be accomplished with either an adaptive or constant algorithm. An adaptive algorithm reads information from the unit and adapts subsequent programming steps to achieve the fastest possible programming time for that unit. Because some in-circuit testers cannot support an adaptive algorithm, programming is accomplished with a constant algorithm.
- CPLDs are used as Local Area Network (LAN) controllers, cache memory controllers, graphics controllers, Universal Asynchronous Receiver/Transmitters (UARTs), and in general in applications that require large quantities of AND/OR gates but not a large number of flip flops. With in-system programmable CPLDs it is possible to reconfigure hardware without powering-down the system.

- Field Programmable Gate Arrays (FPGAs) are similar to PLDs, but whereas PLDs are generally limited to hundreds of gates, FPGAs support thousands of gates. They are especially popular for prototyping integrated circuit designs.
- Field Programmable Gate Arrays (FPGAs) are divided into two major categories: SRAM-based FPGAs and Antifuse-based FPGAs. There is a wide range of FPGAs provided by many semiconductor vendors including Xilinx, Altera, Atmel, and Lattice. Each manufacturer provides each own unique architecture.
- In a typical FPGA, a logic element usually consists of one or more RAM-based n-input look-up tables (LUTs) where n is a number between three and six, and one or more flip-flops. LUTs are used to implement combinational logic. Some FPGAs, like the Xilinx Virtex families, supply on-chip block RAM.
- Quantization occurs if the number of fractional bits is insufficient to represent the fractional portion of a value. We can choose either to truncate by discarding the bits to the right of the least significant bit, or to round to the nearest representable value or to the value farthest from zero if there are two equidistant nearest representable values. A positive double-precision number will have different output depending on whether truncation or rounding is employed. An unsigned double-precision number will also have different output depending on whether truncation or rounding is employed.
- An overflow occurs when a value lies outside the specified range. The user can choose to flag an overflow, or to saturate to the largest positive (or maximum negative value), or to wrap the value, that is, discard any significant bits beyond the most significant bit in the fixed point number.
- CPLDs are typically faster and have more predictable timing than FPGAs. However, FPGAs are generally more dense and contain more flip-flops and registers than CPLDs. Operating speed generally decreases as the PLD density increases.
- Applications of CPLDs and FPGAs include, but not limited to, coprocessors for high speed designs, finite impulse response (FIR) filters, fast Fourier transforms (FFT), and discrete cosine transforms (DCT) that are required for video compression and decompression, encryption, convolution, and forward error correction (FEC) The most commonly used high level languages are Verilog and VHDL. Most manufacturers provide free integrated development environments (IDEs) and tools.

11.9 Exercises

1. A combinational circuit is defined as the functions

$$F_1(A, B, C) = \Sigma(1, 2, 4, 7)$$
$$F_2(A, B, C) = \Sigma(3, 5, 6, 7)$$

Derive the PLA program table for this circuit.

2. A combinational circuit is defined as the functions

$$F_1(A, B, C) = \Pi(0, 2, 3, 5, 6)$$
$$F_2(A, B, C) = \Pi(0, 1, 2, 4, 7)$$

Derive the PLA program table for this circuit.

3. Draw the schematic and derive the equations for a 24-bit adder using 3-bit stages. Express the equations in both Boolean form and in ABEL.

4. As we know, negative numbers are normally stored in twos-complement form. Define the Xilinx *n*-bit fixed point format of the following binary number which is expressed in twos complement form, and compute the equivalent signed decimal number.

1	1	0	0	0	1	1	0	1	0	1	1

5. What format should be used to represent a signal that has been quantized to 12 bits and has a maximum value of +1 and a minimum value of –1 ?

6. What format should be used to represent a signal that has been quantized to 10 bits and has a maximum value of 0.8 and a minimum value of 0.2 ?

7. What format should be used to represent a signal that has been quantized to 11 bits and has a maximum value of 278 and a minimum value of –138 ?

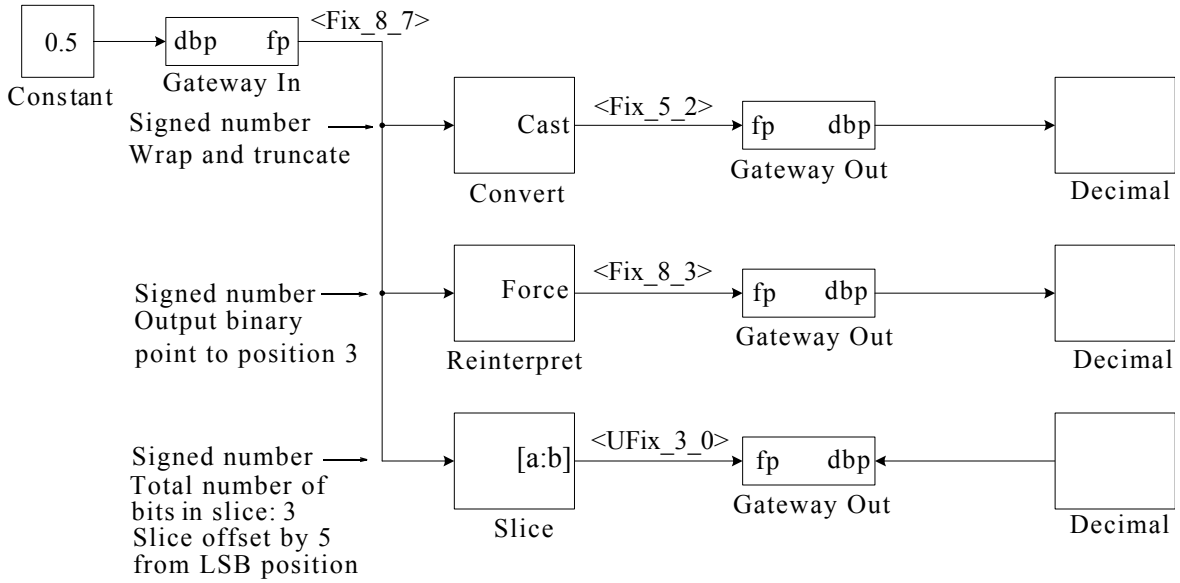
8. Perform the operation

<Fix_12_9> + <Fix_8_3>

9. Perform the operation

<Fix_8_7> × <UFix_8_6>

10. For the block diagram below, determine the binary number at the output of Gateway In, the binary number at the outputs of the Convert, Reinterpret, and Slice blocks, and the outputs of the three Gateway Out blocks. For brevity, “double” indicates double precision. Insert the decimal equivalent values into the blank 3 blocks on the right.


0.5
Constant
dbp fp
Gateway In
<Fix_8_7>
Signed number
Wrap and truncate
Cast
Convert
<Fix_5_2>
fp dbp
Gateway Out
Decimal
Signed number
Output binary
point to position 3
Force
Reinterpret
<Fix_8_3>
fp dbp
Gateway Out
Decimal
Signed number
Total number of
bits in slice: 3
Slice offset by 5
from LSB position
[a:b]
Slice
<UFix_3_0>
fp dbp
Gateway Out
Decimal

11.10 Solutions to End-of-Chapter Exercises

1.

$$F_1(A, B, C) = \Sigma(1, 2, 4, 7) = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$$

$$F_2(A, B, C) = \Sigma(3, 5, 6, 7) = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

The K-maps for F_1, $\overline{F}_1$, F_2, and $\overline{F}_2$ are shown below.

A \ BC	00	01	11	10
0		1		1
1	1		1	

$F_1 = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$

A \ BC	00	01	11	10
0			1	
1		1	1	1

$F_2 = AB + AC + BC$

A \ BC	00	01	11	10
0	0		0	
1		0		0

$\overline{F}_1 = \overline{A}\overline{B}\overline{C} + \overline{A}BC + AB\overline{C} + \overline{A}B\overline{C}$

A \ BC	00	01	11	10
0	0	0		0
1	0			

$\overline{F}_2 = \overline{A}\overline{B} + \overline{A}\overline{C} + \overline{B}\overline{C}$

Examination of F_1, $\overline{F}_1$, F_2, and $\overline{F}_2$ reveals that no matter which combination we select, we will have seven distinct product terms. So, we choose F_1 and F_2 and the PLA program table is shown below.

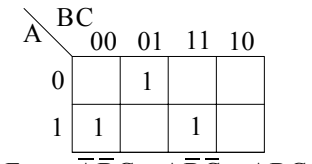
	Product Term	Inputs			Outputs	
		A	B	C	F_1	F_2
AB	1	1	1	—	—	1
AC	2	1	—	1	—	1
BC	3	—	1	1	—	1
$\overline{A}\overline{B}C$	4	0	0	1	1	—
$\overline{A}B\overline{C}$	5	0	1	0	1	—
$A\overline{B}\overline{C}$	6	1	0	0	1	—
ABC	7	1	1	1	1	—
					T	T	T/C

2.

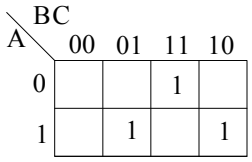
$$F_1(A, B, C) = \Pi(0, 2, 3, 5, 6) = \Sigma(1, 4, 7) = \overline{A}\overline{B}C + A\overline{B}\overline{C} + ABC$$

$$F_2(A, B, C) = \Pi(0, 1, 2, 4, 7) = \Sigma(3, 5, 6) = \overline{A}BC + A\overline{B}C + AB\overline{C}$$

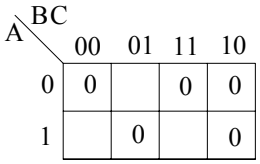
The K-maps for F_1, $\bar{F}_1$, F_2, and $\bar{F}_2$ are shown below.



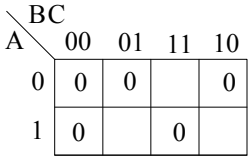
$F_1 = \bar{A}\bar{B}C + A\bar{B}\bar{C} + ABC$



$F_2 = \bar{A}BC + A\bar{B}C + AB\bar{C}$



$\bar{F}_1 = \bar{A}\bar{C} + \bar{A}B + B\bar{C} + A\bar{B}C$



$\bar{F}_2 = \bar{A}\bar{C} + \bar{A}\bar{B} + \bar{B}\bar{C} + ABC$

The simplest combination is obtained from F_1 and $\bar{F}_2$ and the PLA program table is shown below.

	Product Term	Inputs			Outputs	
		A	B	C	F_1	F_2
$\bar{A}\bar{B}$	1	0	0	—	—	1
$\bar{A}\bar{C}$	2	0	—	0	—	1
$\bar{B}\bar{C}$	3	—	0	0	—	1
$\bar{A}\bar{B}C$	4	0	0	1	1	—
$A\bar{B}\bar{C}$	5	1	0	0	1	—
ABC	6	1	1	1	1	1
					T	C	T/C

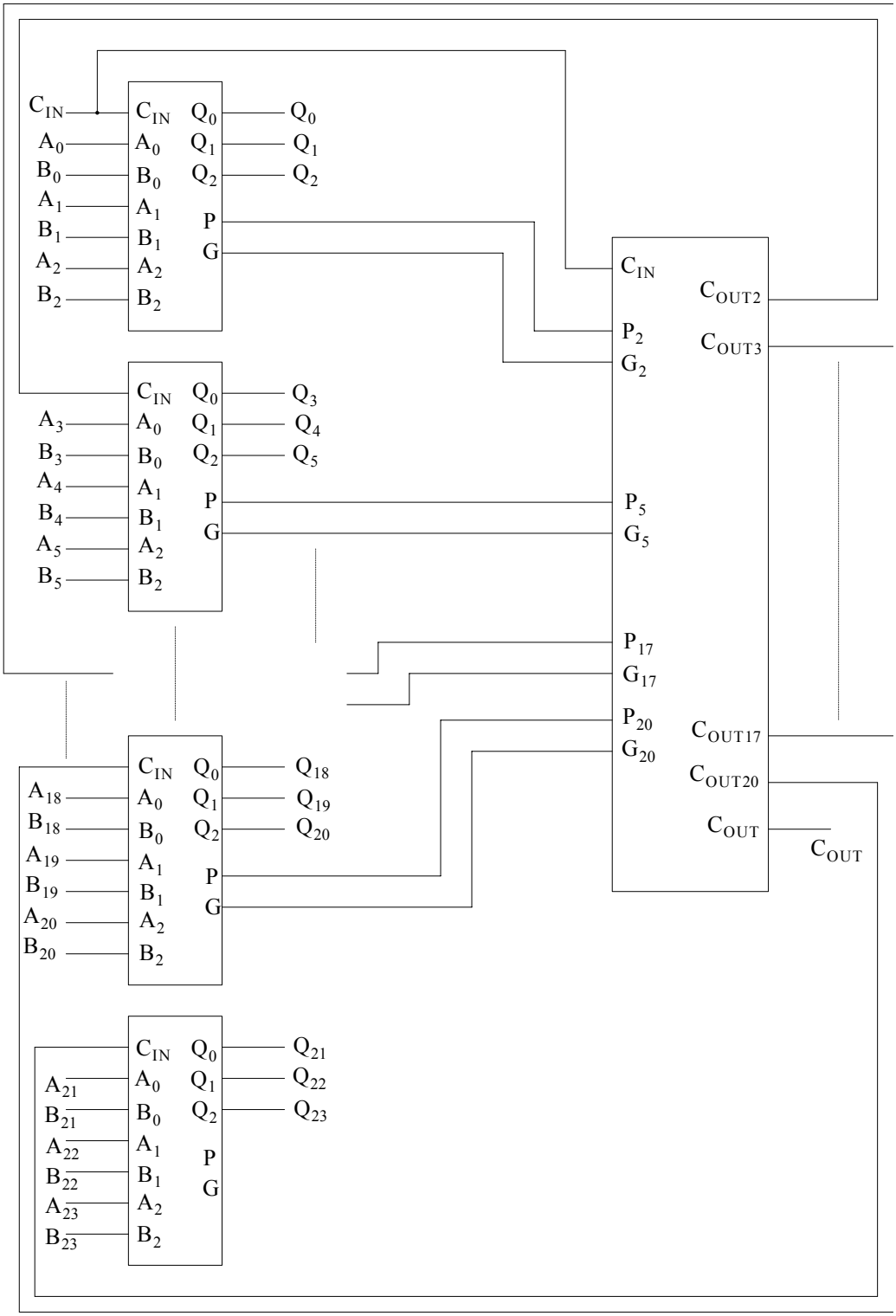
3. The figure below is a 24-bit adder using 3-bit stages. The equations for S_0 and S_1 are the same as the two-bit adder in Example 11.3, that is,

$$S_0 = A_0 \oplus (B_0 \oplus C_{IN0}) = A_0 \oplus (B_0\bar{C}_{IN0} + \bar{B}_0C_{IN0})$$

```
SUM0 = A0 $ ((B0 & !CIN0) # (!B0 & CIN0))
```

$$S_1 = A_1 \oplus [(\bar{A}_0\bar{B}_0B_1 + \bar{A}_0B_1\bar{C}_{IN0} + \bar{B}_0B_1\bar{C}_{IN0}) + (A_0B_0\bar{B}_1 + A_0\bar{B}_1C_{IN0} + B_0\bar{B}_1C_{IN0})]$$

```
SUM1 = A1 $ ((!A0 & !B0 & B1) # (!A0 & B1 & !CIN0) # (!B0 & B1 & !CIN0)
            # (A0 & B0 & !B1) # (A0 & !B1 & CIN0) # (B0 & !B1 & CIN0))
```


C_{IN}
A_0
B_0
A_1
B_1
A_2
B_2
Q_0
Q_1
Q_2
P
G
A_3
B_3
A_4
B_4
A_5
B_5
Q_3
Q_4
Q_5
A_{18}
B_{18}
A_{19}
B_{19}
A_{20}
B_{20}
Q_{18}
Q_{19}
Q_{20}
A_{21}
B_{21}
A_{22}
B_{22}
A_{23}
B_{23}
Q_{21}
Q_{22}
Q_{23}
C_{OUT2}
C_{OUT3}
P_2
G_2
P_5
G_5
P_{17}
G_{17}
P_{20}
G_{20}
C_{OUT17}
C_{OUT20}
C_{OUT}

The equations for the sum and S_2 are:

$$S_2 = A_2 \oplus [(\overline{A}_1\overline{B}_1B_2) + (\overline{A}_0\overline{B}_0\overline{B}_1B_2) + (\overline{A}_0\overline{B}_1\overline{C}_{IN0}B_2) + (\overline{B}_0\overline{B}_1\overline{C}_{IN0}B_2) + (\overline{A}_0\overline{A}_1\overline{B}_0B_2) + (\overline{A}_0\overline{A}_1\overline{C}_{IN0}B_2) + (\overline{A}_1\overline{B}_0\overline{C}_{IN0}B_2) + (A_1B_1\overline{B}_2) + (A_0B_0B_1\overline{B}_2) + (A_0B_1C_{IN0}\overline{B}_2) + (B_0B_1C_{IN0}\overline{B}_2) + (A_0A_1B_0\overline{B}_2) + (A_0A_1C_{IN0}\overline{B}_2) + (A_1B_0C_{IN0}\overline{B}_2)]$$

```
SUM2 = A2 $ ((!A1 & !B1 & B2) # ( !A0 & !B0 & !B1 & B2 )
       # (!A0 & !B1 & !CIN0 & B2) # (!B0 & !B1 & !C IN0 & B2)
       # (!A0 & !A1 & !B0 & B2) # (!A0 & !A1 & !C IN0 & B2)
       # (!A1 & !B0 & !CIN0 & B2) # (A1 & B1 & !B2)
       # (A0 & B0 & B1 & !B2) # (A0 & B1 & C IN0 & !B2)
       # (B0 & B1 & CIN0 & !B2) # (A0 & A1 & B0 & !B2)
       # (A0 & A1 & CIN0 & !B2) # (A1 & B0 & CIN0 & !B2)
```

The generated equations for G_0 and G_1 are the same as in Example 11.3, that is,

$$G_0 = A_0B_0$$

$$G_1 = A_1B_1 + A_0B_0A_1 + A_0B_0B_1$$

```
G0 = A0 & B0
G1 = A1 & B1 # A0 & B0 & A1 # A0 & B0 & B1
```

The generated equation for G_2 is satisfied when:

(1) both A_2 and B_2 are a logical 1 or

(2) at least one of the addends is a logical 1 and there is a carry being generated in the middle stage; or

(3) at least one of the addends is a logical 1 and there is a carry being generated in the first stage and the middle stage propagates the carry to the third stage. Thus,

$$G_2 = A_2B_2 + A_2A_1B_1 + A_2A_1A_0B_0 + A_2B_1A_0B_0 + B_2A_1B_1 + B_2A_1A_0B_0 + B_2B_1A_0B_0$$

```
G2 = (A2 & B2) # (A2 & A1 & B1) # (A2 & A1 & A0 & B0)
     # (A2 & B1 & A0 & B0) # (B2 & A1 & B1)
     # (B2 & A1 & A0 & B0) # (B2 & B1 & A0 & B0)
```

As in Example 11.3, for the propagated equation we can obtain a simpler expression if we consider the case where P_1 is logical 0. This occurs when both A and B are logical 0 in one stage of the three-bit adder. Thus,

$$\bar{P}_1 = \overline{(A_0 + B_0) \cdot (A_1 + B_1) \cdot (A_2 + B_2)} = \bar{A}_0\bar{B}_0 + \bar{A}_1\bar{B}_1 + \bar{A}_2\bar{B}_2$$

```
!P1 = (!A0 & !B0) # (!A1 & !B1) # (!A2 & !B2)
```

Note: We can derive the four-bit equations for implementing a 24-bit adder but the number of product terms would probably exceed the CPLD capacity.

4.

1	1	0	0	0	1	1	0	1	0	1	1

This binary word is in twos complement form and thus represents a negative (signed) binary number. It is 12 bits long, 5 of which are fractional. Therefore,

$$\text{Format} = \langle \text{Fix\_12\_5} \rangle$$

Its true form is found by taking the twos complement. Thus, in true form

0	0	1	1	1	0	0	1	0	1	0	1

Its decimal equivalent is

$$2^4 + 2^3 + 2^2 + 2^{-1} + 2^{-3} + 2^{-5} = 16 + 8 + 4 + \frac{1}{2} + \frac{1}{8} + \frac{1}{32} = 28.65625$$

and since this is a negative number, its decimal value is -28.65625 .

5.

$$+1 \rightarrow 01.0000000000$$
$$-1 \rightarrow 11.0000000000$$

$$\text{Format} = \langle \text{Fix\_12\_10} \rangle$$

6.

$$(0.8)_{10} \approx (.1100110011)_2 = (.79998046875)_{10}$$
$$(0.2)_{10} \approx (.0011001101)_2 = (.199201953125)_{10}$$

These are unsigned values so we should use the format <UFix_10_10>

7.

$$(278)_{10} = (0100010110.0)_2$$
$$(-138)_{10} = (1101110110.0)_2$$

These are signed values so we should use the format <Fix_11_1>

8.

$$\langle \text{Fix\_12\_9} \rangle + \langle \text{Fix\_8\_3} \rangle$$

$$\begin{array}{rl} \text{<Fix\_12\_9>} \rightarrow & \text{XXX.XXXXXXXXX} \\ \text{<Fix\_8\_3>} \rightarrow & \text{XXXXX.XXX} \\ \hline & \text{XXXXXX.XXXXXXXXX} \end{array} +$$

The extra bit in the msb position of the sum accounts for the possibility that there is a carry out from the 2^2 position. Thus,

$$\text{<Fix\_12\_9>} + \text{<Fix\_8\_3>} = \text{<Fix\_15\_9>}$$

9.

$$\begin{array}{rl} \text{<Fix\_8\_7>} \rightarrow & \text{X.XXXXXXX} \\ \text{<Fix\_8\_6>} \rightarrow & \text{XX.XXXXXX} \\ \hline & \text{XXX.XXXXXXXXXXXXX} \end{array} \times$$

Multiplication of a signed number with an unsigned number will yield a signed number. Thus,

$$\text{<Fix\_8\_7>} \times \text{<UFix\_8\_6>} = \text{<Fix\_16\_3>}$$

10.

The input number to *Gateway In* block is assumed to be double and the output is given as format <Fix_8_7>. Thus in binary, the input is as shown in the first block below. It is specified that the output of the *Convert block* is <Fix_5_2>; therefore, the output is as shown in the second block below. The *Reinterpret block* output is specified as <Fix_8_3>; therefore, the output is as shown in the third block below. The Slice block output is specified as <UFix_3_0>; therefore, the output is as shown in the last block below.

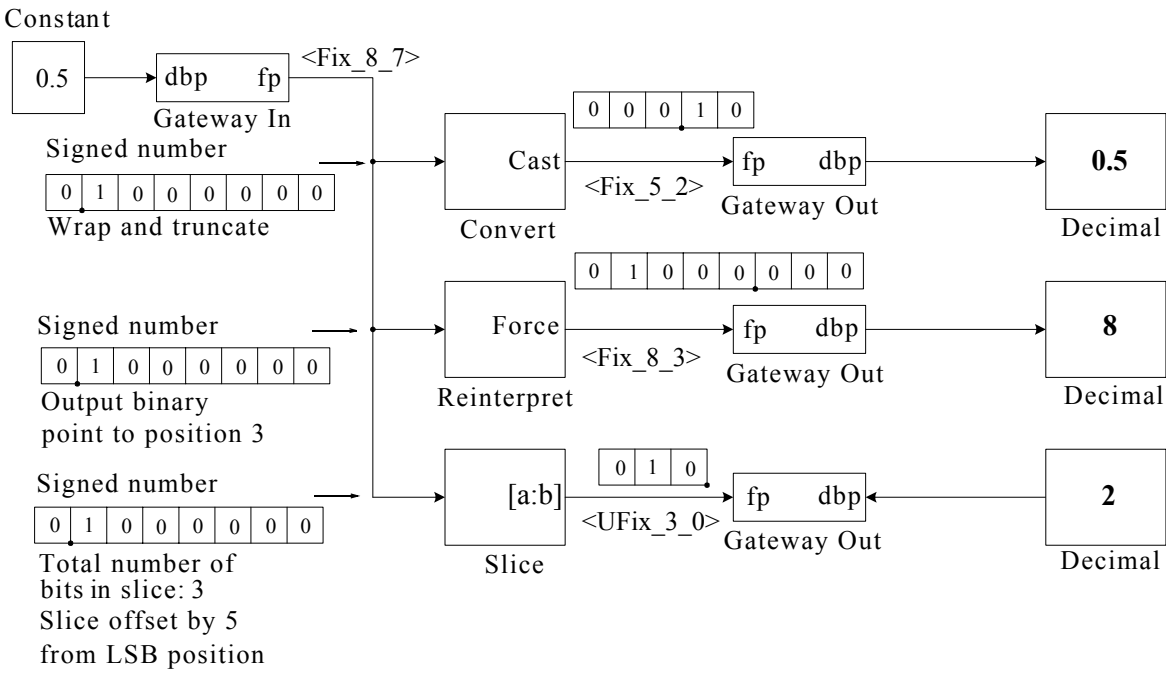
Number input <Fix_8_7>	0	1	0	0	0	0	0	0

Convert output <Fix_5_2>	0	0	0	1	0

Reinterpret output <Fix_8_3>	0	1	0	0	0	0	0	0

Slice output <UFix_3_0>	0	1	0

With these values, the block diagram is as shown below.


Constant
0.5
dbp fp
<Fix_8_7>
Gateway In
Signed number
0 1 0 0 0 0 0 0
Wrap and truncate
Cast
Convert
0 0 0 1 0
<Fix_5_2>
fp dbp
Gateway Out
0.5
Decimal
Signed number
0 1 0 0 0 0 0 0
Output binary
point to position 3
Force
Reinterpret
0 1 0 0 0 0 0 0
<Fix_8_3>
fp dbp
Gateway Out
8
Decimal
Signed number
0 1 0 0 0 0 0 0
Total number of
bits in slice: 3
Slice offset by 5
from LSB position
[a:b]
Slice
0 1 0
<UFix_3_0>
fp dbp
Gateway Out
2
Decimal

Appendix A

Introduction to ABEL Hardware Description Language

This appendix provides a brief overview of the Advanced Boolean Equation Language (ABEL) which is an industry-standard Hardware Description Language (HDL) used in Programmable Logic Devices (PLDs).

A.1 Introduction

The Advanced Boolean Equation Language (ABEL) is an easy-to-understand and use programming language that allows the user to describe the function of logic circuits. It is now an industry-standard allows you to enter behavior-like descriptions of a logic circuit. Developed by Data I/O Corporation for Programmable Logic Devices (*PLDs*), ABEL is now an industry-standard *Hardware Description Language (HDL)*. An example of how ABEL is used, was given in an example in Chapter 11. There are other hardware description languages such as the *VHSIC** *Hardware Description Language (VHDL)* and *Verilog*. ABEL is a simpler language than VHDL and Verilog. ABEL can be used to describe the behavior of a system in a variety of forms, including logic equations, truth tables, and state diagrams using C-like statements. The ABEL compiler allows designs to be simulated and implemented into PLDs such as PALs, CPLDs and FPGAs.

We will not discuss ABEL in detail. We will present a brief overview of some of the features and syntax of ABEL. For more advanced features, the reader may refer to an ABEL manual or the Xilinx on-line documentation.

A.2 Basic Structure of an ABEL Source File

An ABEL source file consists of the following elements:

- Header: including Module, Options, and Title
- Declarations: Pin, Constant, Node, Sets, States, Library
- Logic Descriptions: Equations, Truth-table, State_diagram
- Test Vectors: Test_vectors
- End

Keywords (words recognized by ABEL as commands, e.g. **goto**, **if**, **then**, **module**, etc.) are not case sensitive. User-supplied names and labels (identifiers) can be uppercase, lowercase or mixed-case, but are case-sensitive, for instance `output1` is different from `Output1`.

* *VHSIC is an acronym for Very High Speed Integrated Circuits.*

Example A.1

A source file is given below.

module *module name*

[**title** *string*] (optional)

[deviceID **device** deviceType;] (optional)

pin declarations (optional)

other declarations (optional)

comments (optional)

equations

equations

[**Test_Vectors**] (optional)

test vectors

end *module name*

Using the structure of this source file, write a source file that implements a 4-bit full adder.

Solution:

```
module circuit_05;
title ' EE124 Lab Experiment 05'
Four_bit_adder device '74HC283';
" input pins
A1, B1, A2, B2, A3, B3, A4, B4, CIN pin 5, 6, 3, 2, 14, 15, 12,
11, 7;

 " output pins
 S1, S2, S3, S4, COUT pin 4, 1, 13, 10 9 istype 'com';
equations
S1 = A1 $ B1 $ CIN ;
COUT1 = (A1 & B1) # (A1 & CIN) # (B1 & CIN) ;
S2 = A2 $ B2 $ COUT1 ;
COUT2 = (A2 & B2) # (A2 & COUT1) # (B2 & COUT1) ;
S3 = A3 $ B3 $ COUT2 ;
COUT3 = (A3 & B3) # (A3 & COUT2) # (B3 & COUT2) ;
S4 = A4 $ B4 $ COUT3 ;
COUT = (A4 & B4) # (A4 & COUT3) # (B4 & COUT3) ;
```

```
end circuit_05;
```

A brief explanation of the statements follows.

module: is a module statement followed by the module name (identifier) `circuit_05.` The module statement is mandatory.

title: is optional but in this example here it is used to identify the project. The title name must be between single quotes. It is ignored by the compiler but is recommended for documentation purposes.

string: is a series of ASCII characters enclosed by single quotes. Strings are used for **title**, **options** statements, and in pin, node and attribute declarations.

device: this declaration is optional and associates a device identifier with a specific programmable logic device. The device statement must end with a semicolon. The format is as follows:

device_id **device** 'real_device';

For this example it is: `Four_bit_adder` **device** `'74HC283';`

comments: comments can be inserted anywhere in the file and begin with a double quote and end with another double quote or the end of the line, whatever comes first. For this example, `" input pins and " output pins` are comments.

equations: We use this keyword to write the Boolean expressions using the following operators:

TABLE A.1 Operators used with Boolean expressions in ABEL

Operator	Description
&	AND
#	OR
!	NOT
$	XOR
!$	XNOR

end: End of the source file whose name was specified in module. For this example it is

```
end circuit_05;
```

A.3 Declarations

In this section we provide a general description of the ABEL declarations listed below.

module *module name*

[**title** *string*]

[deviceID **device** deviceType;]

comments: Comments can be inserted anywhere in the file and begin with a double quote and end with another double quote or the end of the line, whatever comes first.

pin declarations

other declarations

equations

equations

[**Test_Vectors**]

test vectors

end *module name*

module: A source files starts with a module statement followed by a module name (identifier). Large source files can be broken down to a group of smaller modules with their own individual title, equations, end statements, etc.

title: The title is optional and it is ignored by the complier but it is recommended for easy identification. If used, its name must be written between single quotes.

string: Strings, a series of ASCII characters, are used for **title**, **options** statements, and in **pin**, **node** (defined below), and attribute declarations. Strings must be enclosed in single quotes.

device: Device is another optional declaration and if used, it associates a device identifier with a specific programmable logic device. The device statement must end with a semicolon. The format is as follows:

device_id **device** 'real_device';

Example: Quad_2_input_Multiplexer **device** '74HC157';

pin: Pin declarations tell the compiler which symbolic names are associated with the external pins of the device. For active Low, we use the (!) symbol meaning that the signal will be inverted. The `istype` is an optional attribute assignment for a pin such as `com` to indicate that the output is a combinational signal or `reg` for a clocked signal (registered with a flip flop). The `istype` attribute is only used for output pins. Thus, for active High the format is:

```
pin_id pin [pin#] [istype 'attributes'] ;
```

and for active Low the format is:

```
[!]pin_id pin [pin#] [istype 'attributes'] ;
```

We can specify more than one pin per line with the following format:

```
pin_id , pin_id, pin_id pin [pin#, [pin#, [pin#]]] [istype
'attributes'];
```

Examples:

```
IN1, IN2, A1, B1 pin 2, 3, 4, 5;
OUT1 pin 7 istype 'reg';
ENABLE pin 10;
!Chip_select pin 12 istype 'com';
!S0..!S6 pin istype 'com';
```

It is not mandatory to specify the pin numbers, but it is a good practice for identification purposes.

node: The node declarations have the same format as the pin declaration. Nodes are internal signals, that is, they are not connected to external pins.

Example:

```
tmp1 node [istype 'com'];
```

Other declarations allow us to define constants, sets, macros and expressions that can simplify the program. As an example a constant declaration has the following format:

id [, id],... = expr [, expr].. ;

Examples:

```
A = 15;
B=3*8;
ADDRESS = [0,1,14];
PRODUCT = B & C;
```

The following lines are examples of a vector notation and they are equivalent.

```
A = [A5, A4, A3, A2, A1, A0];
A = [A5..A0];
```

The last two equations are equivalent. The use of ".." in the last line above is convenient way to specify a range. Thus, whenever we use A in an equation, it will refer to the vector [A5, A4, A3, A2, A1, A0].

A.4 Numbers

ABEL recognizes numbers in binary, octal, decimal and hexadecimal formats. The default base is decimal, that is, when no symbol is specified, it is assumed to be in the decimal base. Table A.2 shows the symbols (upper or lower case allowed) we can use to specify the base. We can replace the default base with another base using the directive **@Radix** as discussed in Section A.5.

TABLE A.2 Number bases used in ABEL

Name	Base	ABEL Symbol
Binary	2	^b
Octal	8	^o
Decimal	10	^d (default)
Hexadecimal	16	^h

Example A.2

The first column in Table A.3 shows the numbers specified in ABEL. Provide the equivalent decimal numbers in the second column.

TABLE A.3 Table for Example A.2

Specified in ABEL	Decimal Equivalent
^b10111	
^o110111	
45	
^hF3A5	

Solution:

The decimal equivalents are shown in Table A.4.

TABLE A.4 Table for Example A.2 - Solution

Specified in ABEL	Decimal Equivalent
^b10111	23
^o110111	67
45	45
^hDB0A	3,504,625

A.5 Directives

Directives allow manipulation of the source file and processing. Directives can be placed anywhere in the source file.

A.5.1 The @alternate Directive

Syntax : **`@alternate`**

The **`@alternate`** directive allows us to specify an alternate set of operators. Please refer to Table A.4 in Section A.7.1, for a list of alternate operators. As shown in Table A.4, we cannot

use (+) for addition, (*) for multiplication, and (/) for division because they represent the OR, AND and NOT logical operators respectively in the alternate set. The standard operator still work when **@alternate** is in effect. The alternate operators remain in effect until the @STANDARD directive is used or the end of the module is reached.

A.5.2 The @radix Directive

Syntax : **@radix** expr ;

where expr is a valid expression that produces the number 2, 8, 10 or 16 to indicate a new default base number. The **@radix** directive changes the default base. The default is base 10 (decimal). The newly-specified default base stays in effect until another **@radix** directive is issued or until the end of the module is reached. Note that when a new **@radix** is issued, the specification of the new base must be in the current base format.

Example A.3

Use the **@radix** directive to convert the decimal number 3,504,625 to its equivalent hexadecimal. Use this directive again to convert from hexadecimal back to decimal.

Solution:

@radix 16; "change default base to hexadecimal

@radix DB0A; "change back from hexadecimal to decimal

A.5.3 The @standard Directive

Syntax : **@standard**

The **@standard** directive resets the operators to the ABEL-HDL standard. As stated in subsection A.5.1, the alternate set is chosen with the **@alternate** directive.

A.6 Sets

A *set* is a collection of variables or constants that allows us to reference a group by one name. A set provides a convenient way simplify logic expressions. Any operation that is applied to a set is applied to each element of the set. A set is separated by commas or the range operator (..) and must be enclosed in square brackets. For example, suppose a set is defined as

```
[A0, A1, A2, A3, A4]
```

We can increment this range with the set

```
[A0..A7]
```

As another example, suppose that a set is defined as

```
[a0..a8]
```

We can decrement this range with the set

```
[a4..a0]
```

We can also redefine the range

```
[A0..A7]
```

as

```
[A8..A15]
```

A set can also have a range within a larger set. For instance,

```
[A0..A7, B2..B5]
```

The options above apply also to active Low variables, for instance

```
[!B8..!B0]
```

A set of the form

```
[X0, Y]
```

is invalid. However, if

```
Y = [Y0..Y3]
```

we can express it in valid for as

```
[X0, Y0..Y3]
```

A.6.1 Indexing or Accessing a Set

We can access elements within a set using *indexing*. We must use numerical values to indicate the set index. The numerical values refer to the bit position in the set starting with 0 for the least significant bit of the set. Some examples follow where the operator := in lines 3 and 4 below is an assignment operator. We will discuss assignment operators in Subsection A.7.4.

```
A1 = [A9..A0]; "set declaration
B2 = [B3..B0]; "set declaration
B2 := A1[5..0]; "makes X2 equal to [A5, A4, A3, A2, A1, A0]
B2 := A1[8..5]; "makes X2 equal to [A8, A7, A6, A5]
```

In our previous discussion, we have used the operator (=) as an assignment operator. In ABEL, the assignment operator is used in equations whereas the equality operator is denoted as (==).

To access one element as an output in a set, we use the following syntax:

```
OUTPUT = (B[2] == 1);
```

The equality sign (==) gives a logical 1 or a logical 0 depending on the output being True or False.

A.6.2 Set Operations

Set Operations are operations applied to a set. Unless we use parentheses to specify precedence, operators with the same priority are performed from left to right. Some simple examples follow to illustrate set operations.

Example A.4

Let

```
VOUT = [V2, V1, V0]; "Declaration of an output voltage set
```

What is VOUT if:

a. `VOUT = [1,0,1] & [0,1,1];`

b. `VOUT = [1,0,1] # [0,1,1];`

Solution:

a. Each element of the first set is ANDed with the corresponding element of the second set, and thus

```
VOUT = [1 & 0, 0 & 1, 1 & 1]=[0, 0, 1]
```

b. Each element of the first set is ORed with the corresponding element of the second set, and thus

```
VOUT =  [1 # 0, 0 # 1, 1 # 1] = [1, 1, 1]
```

Example A.5

What is `!V` if `V = [V1, V2, V3];` ?

Solution:

```
!V = [!V1, !V2, !V3];
```

The statement

```
[A,B]= C & D;
```

is equivalent to the two statements

```
A= C & D;

B= C & D;
```

Example A.6

What is the statement

```
[X1,Y1] = [A1,A2] & [B2,B3]
```

equivalent to?

Solution:

```
[X1,Y1] = [A1 & B2, A2 & B3];
```

Thus,

```
[X1] = [A1 & B2];
```

and

```
[Y1] = [A2 & B3];
```

Example A.7

What is the statement

```
VIN = A1 & [B1,C1,D1]
```

equivalent to?

Solution:

```
VIN = A1 & B1, A1 & C1, A1  & D1];
```

Consider the statement

```
VOUT = 3 & [V1, V2, V3, V4];
```

This is a Boolean expression and the constant 3 must be expressed in binary form. Also, because the bracketed term contains 4 variables, the constant 3 must be padded with 2 leading 0s, that is, it must be expressed as 0011. Therefore,

```
VOUT = 0011 & [V1, V2, V3, V4];
```

or

```
VOUT = [0 & V1, 0 & V2, 1 & V3, 1 & V4]; = [V3, V4]
```

Example A.8

Consider the statement

```
VOUT = 3 & [!V1, !V2, !V3, V4];
```

This is a Boolean expression and the constant 3 must be expressed in binary form. Thus,

```
VOUT = 0011 & [!V1, !V2, !V3, V4];
```

But this cannot be true because `V3` is complemented, that is, `!V3 = 0`. Therefore, the binary

number 0011 is truncated to 0001 and thus the expression

```
VOUT = 3 & [!V1, !V2, !V3, V4];
```

is the same as

```
VOUT = 1 & [!V1, !V2, !V3, V4];
```

Example A.9

What is the statement

```
[X3, X2, X1, X0] = 2
```

equivalent to?

Solution:

As in Example A.7, the constant 2 is converted into binary and padded with zeros, that is, 0010. Then,

```
X3 = 0     X2 = 0     X1 = 1     X0 = 0
```

Sets are also being used with logic expressions. The logic expressions are very helpful when working with a large number of variables, such as a 16-bit address For instance, we can use sets to express the logic expression

```
Input = A3 & !A2 & !A1 & A0
```

as the declaration

```
VIN = [A3, A2, A1, A0];
```

Now, we can use the following equation to specify VIN as follows:

```
Input = VIN == [1, 0, 0, 1]
```

Thus, if A3 = 1, A2 = 0, A1 = 0, and A0 =1, the expression `VIN == [1, 0, 0, 1]` is logic 1 (True) and `Input` will also be logic 1. We can also use the equation

```
Input = VIN == 9;
```

where $(9)_{10} = (1001)_2$.

A.7 Operators

There are four basic types of operators: logical, arithmetic, relational and assignment.

A.7.1 Logical Operators

The *logical operators* are shown in Table A.5. Operations are performed bit by bit. We can use the alternate operators shown on the right column of Table A.5 with the **`@alternative`** directive.

TABLE A.5 Default and Alternate Logic Operators

Default Operator	Description	Alternate Operator
!	NOT (inversion)	/
&	AND	*
#	OR	+
$	XOR	:+:
!$	XNOR	:*:

A.7.2 Arithmetic Operators

The table below gives the *arithmetic operators*. Note that the last four operators are not allowed with sets. The minus (–) sign can have different meanings: used between two operands it indicates subtraction (or adding the twos complement), while used with one operator it indicates the twos complement. The arithmetic operators *, /, %, <<, and >> shown on the last 5 rows of Table A.6 cannot be used with sets.

TABLE A.6 Arithmetic Operators

Operator	Description	Examples
–	Twos complement	–A
–	Subtraction	A1–B1
+	Addition	C1+D1
*	Multiplication	A*B
/	Integer Division (unsigned numbers)	C/D
%	Modulus (Remainder of Division)	A%B (Remainder of A/B)
<<	Shift Left	A<<B (Shift Left A by B bits)
>>	Shift Right	C>>D (Shift Right C by D bits)

A.7.3 Relational Operators

The *relational operators* are shown in Table A.7. These operators produce a Boolean value of True (logic 1) or False (logic 0).

TABLE A.7 Relational Operators

Operator	Description	Assumptions	Examples
= =	Equal	A=5+7, B=3+9	A = = B (True)
!=	Not equal	C=17, D=12	C!=D (True)
<	Less than	a = 8, b = 4	a < b (False)
<=	Less than or equal to	c = 3, d = 5	c <= d (True)
>	Greater than	e = 11, f = 8	f > e (False)
>=	Greater than or equal to	g = 2, h = 3	h >= g (True)

We recall that a binary number 8 bits long can be represented in the range from –128 (11111111) to +127 (01111111), and a binary number 16 bits long can be represented in the range –32768 (1111111111111111) to +32767 (0111111111111111). The logical true value of –1 in twos complement in a 16 bit word is as 1111 1111 1111 1111.* It is important, however, to remember that *relational operators are unsigned*. Thus, `–1>3` and likewise !0 is the one complement of 0 or 11111111 (8 bits data) which is 255 in unsigned binary. Thus !0 > 7 is true.

Relational expressions are very useful in specifying conditional logical expressions. For instance, if we want X to be equal to Y if A is not equal to B and produce a logic 1 (True) when this condition is satisfied, and to produce a logic 0 (False) otherwise, we can use the statement

```
X = Y !$ (A!=B)
```

A.7.4 Assignment Operators

Assignment operators are those used in equations to assign the value of an expression to output variables. There are two types of assignment operators: *combinational* and *registered*. For a combinational assignment operator the syntax is

```
[!]pin_id pin [pin#] [istype 'com'];
```

and for a registered assignment operator the syntax is

```
[!]pin_id pin [pin#] [istype 'reg'];
```

In a combinational operator the assignment occurs immediately without any delay. However, the registered assignment occurs at the next clock pulse associated with the output. For instance, we can define the output of a flip-flop with either of the following statements:

```
Q1_out pin istype 'reg';

Q1 := D1;
```

The first statement defines the output Q1 of a flip-flop by using the 'reg' as istype (registered output). The second statement assumes that Q1 is the output of a D-type flip flop and the state of this output will be the same as the state of flip-flop D1 at the next clock transition.

A.7.5 Operator Priorities

The precedence (priority) of each operator is as shown in Table A.8 with priority 1 the highest and 4 the lowest. When operators have the same priority, the operations are performed from left to right.

* *This can be verified by taking the ones complement of 1111111111111111 which is 0000000000000000 and add 1 to the lsb.*

TABLE A.8

Priority	Operator	Description
1	–	Negation (twos complement)
1	!	NOT
2	&	AND
2	<<	Shift left
2	>>	Shift right
2	*	Multiplication
2	/	Division (Unsigned)
2	%	Modulus (remainder)
3	+	Addition
3	–	Subtraction
3	#	OR
3	$	XOR
3	!$	XNOR
4	= =	Equal
4	!=	Not equal
4	<	Less than
4	<=	Less or equal
4	>	Greater than
4	>=	Greater or equal

A.8 Logic Description

A logic design can be described in the following way.

- Equations
- Truth Table
- State Description

A.8.1 Equations

We begin the logic description with the word **`equations`**. The equations specify logic expressions using the operators listed in Table A.8, or with the `When-Then-Else` statement. This statement is used for the description of logic functions. The `If-Then-Else` statement is used to describe state progression and it will be discussed in Subsection A.8.3.

The format of the `When-Then-Else` statement is as follows:

```
WHEN condition THEN element=expression;
ELSE equation;
```

or

```
WHEN condition THEN equation;
```

Some simple examples follow.

1.
```
SUM0 = (A0 & !B0) # (!A0 & B0);
```

2.
```
X0 := CLK & A2 & !A1 & !A0;
```

3.
```
WHEN (A1 == B1) THEN Q1 = A2;
  ELSE WHEN (A1 == C1) THEN Q1 = A0;
```

We can use the braces { } to group sections together in blocks. For instance,

```
WHEN (D>C) THEN {Q1 :=D1; Q2 :=D2;}
```

The text in a block can be on one line as above, or it can be expressed in two or more lines. Blocks are used in equations, state diagrams and directives.

A.8.2 Truth Tables

With *truth tables* we use the words **`truth_table`** and the syntax can be written in either of the three expressions below where `->` is used for outputs of combinational circuits, and `:>` is used for outputs of sequential (registered) circuits.

1.
```
TRUTH_TABLE (in_ids -> out_ids)
inputs -> outputs ;
```

2.
```
TRUTH_TABLE (in_ids :> reg_ids)
inputs :> reg_outs ;
```

3
```
TRUTH_TABLE (in_ids :> reg_ids -> out_ids)
inputs :> reg_outs -> outputs ;
```

The first line of a truth table (between parentheses) defines the inputs and the output signals. The following lines show the values of the inputs and outputs. Each line must end with a semicolon. The inputs and outputs can be single signals or sets. When sets are used as inputs or outputs, we use the normal set notation, i.e. signals surrounded by square brackets and separated by commas. A don't care is represented by an `.X.`

Example A.10

Write the truth table for a full adder.

Solution:

We wrote the truth table for a full adder in Chapter 7. In ABEL the first line is

```
TRUTH_TABLE ([ A, B, CIN] -> [SUM, COUT])

     [ 0, 0, 0 ] -> [ 0, 0 ];
     [ 0, 0, 1 ] -> [ 1, 0 ];
     [ 0, 1, 0 ] -> [ 1, 0 ];
     [ 0, 1, 1 ] -> [ 0, 1 ];
     [ 1, 0, 0 ] -> [ 1, 0 ];
     [ 1, 0, 1 ] -> [ 0, 1 ];
     [ 1, 1, 0 ] -> [ 0, 1 ];
     [ 1, 1, 1 ] -> [ 1, 1 ];
```

The truth table above can be simplified if we define the set

$$IN = [A,B];$$

and

$$OUT = [SUM, COUT]$$

Then, the truth table is written as

```
TRUTH_TABLE ( IN -> OUT)

     0 -> 0;
     1 -> 2;
     2 -> 2;
     3 -> 1;
     4 -> 2;
     5 -> 1;
     6 -> 1;
     7 -> 3;
```

Example A.11

The output of an XOR gate is connected to an ON/OFF switch. Write the truth table.

Solution:

Let us denote the switch as SW where the OFF position is represented as logic 0, and the ON position as logic 1, and the inputs to the XOR gate as A and B. Using a short notation as in Example A.10, the truth table is written as follows where `.X.` indicates a don't care.

```
TRUTH_TABLE ([ SW, A, B ] -> OUT)

     [ 0, .X., .X. ] -> [ .X. ];
     [ 1, 0, 0 ] -> 0;
     [ 1, 0, 1 ] -> 1;
     [ 1, 1, 0 ] -> 1;
     [ 1, 1, 1 ] -> 0;
```

Truth tables can also be used to define sequential circuits. In Chapter 8, Example 8.5, we designed a BCD up counter. In the following example, we will implement this counter in ABEL.

Example A.12

Implement a BCD up counter which counts from 0000, 0001, ... to 1001 and back to 0000. Make provisions that the counter will generate an output OUT whenever the counter reaches the state 1001. Assume that the flip flops can be simultaneously reset to 0000 by the reset direct (RD) signal.

Solution:

The state table shown as Table A.9 lists the present and next states where Q_4 is the most significant bit (msb) and Q_1 as the least significant bit (lsb). We can generate the output OUT (combinational signal) by adding a 4-input AND gate that produces a logic 1 output when the input is $Q_4\overline{Q}_3\overline{Q}_2Q_1$.

TABLE A.9 State table for Example A.12

Present State				Next State			
Q_4	Q_3	Q_2	Q_1	Q_4	Q_3	Q_2	Q_1
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

The implementation in ABEL is shown below where have used the dot extensions `.CLK` and `.AR`. We will discuss dot extensions in Subsection A.8.4.

```
module BCD_COUNTER;

CLOCK pin; " input signal

RESET pin; " input signal
```

```
OUT pin istype 'com'; " output signal (combinational)
Q4, Q3, Q2, Q1 pin istype 'reg'; " output signal (registered)

[Q4, Q3, Q2, Q1].CLK = CLOCK; "FF clocked on the CLOCK input
[Q4, Q3, Q2, Q1].AR = RESET; "asynchronous reset by DIRECT RESET

TRUTH_TABLE ([Q4, Q3, Q2, Q1] :> [Q4, Q3, Q2, Q1] -> OUT)

                [ 0 0 0 0 ] :> [ 0 0 0 1 ] -> 0;
                [ 0 0 0 1 ] :> [ 0 0 1 0 ] -> 0;
                [ 0 0 1 0 ] :> [ 0 0 1 1 ] -> 0;
                [ 0 0 1 1 ] :> [ 0 1 0 0 ] -> 0;
                [ 0 1 0 0 ] :> [ 0 1 0 1 ] -> 0;
                [ 0 1 0 1 ] :> [ 0 1 1 0 ] -> 0;
                [ 0 1 1 0 ] :> [ 0 1 1 1 ] -> 0;
                [ 0 1 1 1 ] :> [ 1 0 0 0 ] -> 0;
                [ 1 0 0 0 ] :> [ 1 0 0 1 ] -> 0;
                [ 1 0 0 1 ] :> [ 0 0 0 0 ] -> 1;
            [ 1 0 1 0 ] :> [.X. .X. .X. .X.] -> .X.;
            [ 1 0 1 1 ] :> [.X. .X. .X. .X.] -> .X.;
            [ 1 1 0 0 ] :> [.X. .X. .X. .X.] -> .X.;
            [ 1 1 0 1 ] :> [.X. .X. .X. .X.] -> .X.;
            [ 1 1 1 0 ] :> [.X. .X. .X. .X.] -> .X.;
            [ 1 1 1 1 ] :> [.X. .X. .X. .X.] -> .X.;

end BCD_COUNTER;
```

A.8.3 State Diagram

This subsection describes the state_diagram that contains the state description for the logic design. We can declare symbolic state names in the declaration section, and this makes the reading easier. We will discuss the *state_diagram* syntax and the *If-Then-Else, Goto, Case,* and *With* statements.

In the declaration section, the *state declaration* syntax is

```
state_id [, state_id ...] STATE ;
```

Suppose we have an 8-bit register and we assign the state name REG8 to it. We can associate this state name with the outputs $Q_0, Q_1, \ldots Q_7$ as

$$REG8 = [Q_0 .. Q_7]$$

The syntax for `State_diagram` is as follows:

```
State_diagram state_reg
STATE state_value : [equation;]
[equation;]
:
:
trans_stmt ; ...
```

The word **state_diagram** in the first line above is used to indicate the beginning of a state description.

The STATE word and following statements describe one state of the state diagram and includes a state value or symbolic state name, state transition statement, and an optional output equation. Thus, in the second line, after STATE we may write either

1. state_value: this can be an expression, a value or a symbolic state name of the current state.

2. state_reg: is an identifier that defines the signals that determine the state of the machine. This can be a symbolic state register that has been declared earlier in the declaration section.

The [equation;] in the second line is optional.

The [equation;] in the third line is mandatory; it is an equation that defines the outputs.

In the last line, trans_stmt, can be used with the If-Then-Else, Goto, or Case statements to defines the next state, followed with optional With transition equations.

The **If-Then-Else** statement is used to determine the next state depending on mutually exclusive transition conditions. Its syntax is

```
IF expression THEN state_exp
[ELSE state_exp] ;
```

The state_exp can be a logic expression or a symbolic state name. We must remember that the If-Then-Else statement can only be used in the state_diagram section.* The ELSE† clause is optional. The If-Then-Else statements can be nested with Goto, Case, and With statements. As an example, in the declaration section we first define the state registers:

```
REG8=[Q0..Q7]; " Define 8-bit register

SUM = (A & !B) # (!A & B) ;
SUM0 = [0, 0];
SUM1 = [1, 1];
```

* *We recall from Subsection A.8.1 that the When-Then-Else statement is used with equations.*

† *As stated earlier, words recognized by ABEL as commands, e.g.* ***goto, if, then, module****, are not case-sensitive.*

```
state_diagram REG8
state SUM0: OUT1 = 1;
    if A then S1
        else S0;
state SUM1: OUT2 =1;
    if A then S0
       else S1;
```

The **with** statement is used with the syntax:

```
trans_stmt state_exp WITH equation
[equation ] ... ;
```

where the `trans_stmt` can be `If-then-else`, `Goto`, or a `Case` statement, `state_exp` is the next state, and `equation` is an equation for the machine outputs. This statement can be used with the `If-then-else`, `Goto`, or `Case` statements in place of a simple state expression. The `with` statement allows the output equations to be written in terms of transitions. For example, in the statement

```
if A0 & B0 ==1 then SUM0 with C0=1 else SUM1
```

the output `C0` will be true `if A0 & B0 ==1` is True (logic 1). Any expression after `with` can be an equation and it will be evaluated when the `if` condition is true. The following statement is also valid.

```
if A0 # B0 ==1 then SUM0 with C0=A0 & B0 else SUM1 with A0 != B0
```

The **Case** statement is used with the syntax

```
case expression : state_exp;
[ expression : state_exp; ]
:
endcase ;
```

where `expression` is any valid expression and `state_exp` is an expression indicating the next state. For example,

```
State SUM0:
     case ( A == 0) : SUM1;
          ( A == 1) : SUM0;
   endcase;
```

The `case` statement is used to list a sequence of mutually-exclusive conditions and corresponding next states. The `case` statement conditions must be mutually exclusive, that is, two conditions cannot be true at the same time.

A.8.4 Dot Extensions

We can use *dot extensions* to describe the behavior of a digital circuit more precisely. The extensions are very convenient and provide a means to refer specifically to internal states and nodes associated with a primary condition. Dot extensions are not case sensitive. Dot extensions are used with the syntax

```
signal_name.ext
```

Several dot extensions are listed in Table A.10.

TABLE A.10 Dot extensions

Extension	Description
.ACLR	Asynchronous register reset
.ASET	Asynchronous register preset
.CLK	Clock input to an edge-triggered flip flop
.CLR	Synchronous register reset
.COM	Cominbational feedback from flip flop data input
.FG	Register feedback
.OE	Output enable
.PIN	Pin feedback
.SET	Synchronous register preset
.D	Data input to a D Flip flop
.J	J input to a JK flip-flop
.K	K input to a JK flip-flop
.S	S input to a SR flip-flop
.R	R input to a SR flip-flop
.T	T input to a T flip-flop
.Q	Register feedback
.PR	Register preset
.RE	Register reset
.AP	Asynchronous register preset
.AR	Asynchronous register reset
.SP	Synchronous register preset
.SR	Synchronous register reset

As an example, the statements

```
Q.AR = reset;

[Q1.ar, Q2.ar] = reset;
```

reset the flip flop outputs when `reset` is logic 1. As another example, the statement

```
[LD7..LD0].OE = ACTIVE;
```

will enable the outputs of line drivers `LD7` through `LD0` when `ACTIVE` is logic 1, otherwise the outputs will be in a high-Z state.

A.9 Test Vectors

Test vectors are optional but are useful for providing a means to verify the correct operation of a state. The vectors specify the expected logical operation of a logic device by explicitly giving the outputs as a function of the inputs. Test vectors are used with the syntax

```
Test_vectors [note]
(input [, input ].. -> output [, output ] .. )
[invalues -> outvalues ; ]

:

:
```

As an example, we can use test vectors to verify the outputs of a full adder as follows:

```
  Test_vectors

   ([ A, B, CIN] -> [SUM, COUT])

      [ 0, 0, 0 ] -> [ 0, 0 ];
      [ 0, 0, 1 ] -> [ 1, 0 ];
      [ 0, 1, 0 ] -> [ 1, 0 ];
      [ 0, 1, 1 ] -> [ 0, 1 ];
      [ 1, 0, 0 ] -> [ 1, 0 ];
      [ 1, 0, 1 ] -> [ 0, 1 ];
      [ 1, 1, 0 ] -> [ 0, 1 ];
      [ 1, 1, 1 ] -> [ 1, 1 ];
```

As with truth tables, we can also specify the values for the set with numeric constants as shown below.

```
  Test_vectors

([ A, B, CIN] -> [SUM, COUT])

       0 -> 0;
       1 -> 2;
```

```
0 -> 0;
1 -> 2;
2 -> 2;
3 -> 1;
4 -> 2;
5 -> 1;
6 -> 1;
7 -> 3;
```

Don't cares (.X.), clock inputs (.C.), and symbolic constants are also allowed as shown in the following example.

```
test_vectors

( [CLK, RESET, QA, QB ] -> [ X0, X1, X2] )
  [.X., 1, .X.,.X.] -> [ A0, 0, 0];
  [.C., 0, 0, 1 ]   -> [ B0, 0, 0];
  [.C., 1, 1, 0 ]    ->[ C0, 0, 1];
```

A.10 Property Statements

Property statements allow us to assign device specific statements. For field programmable devices the property statements include

- Slew rates
- Power settings
- Preload values

A.11 Active-Low Declarations

As we've learned, active low signals are defined with a "!" operator. Consider, for example, the declaration

```
!OUT pin istype 'com' ;
```

When this signal is used in a subsequent design description, it will be automatically complemented. As an example consider the following description,

```
module EXAMPLE
A, B pin ;
!OUT pin istype 'com';
equations
OUT = A & !B # !A & B ;
end
```

In this example, the signal OUT is an XOR of A and B, i.e. OUT will be "1" (High, or ON) when

only one of the inputs is "1", otherwise OUT is "0". However, the output pin is defined as !OUT , i.e. as an active-low signal, which means that the pin will go low "0" (Active-low or ON) when only one of the two inputs are "1". We could have obtained the same result by inverting the signal in the equations and declaring the pin to be OUT, as is shown in the following example. This is called explicit pin-to-pin active-low (because one uses active-low signals in the equations).

```
module EXAMPLE
A, B pin ;
OUT pin istype 'com';
equations
!OUT = A & !B # !A & B ;
end
```

Active low can be specified for a set as well. As an example lets define the sets A, B, and C.

```
A = [A2,A1,A0]; "set declaration
B = [B2,B1.B0]; "set declaration
C = [X2,X1.X0];  "set declaration
!C = A & !B # !A & B;
```

The last equation is equivalent to

```
!C0 = A0 & !B0 # !A0 & B0;
!C1 = A1 & !B1 # !A1 & B1;
!C2 = A2 & !B2  # !A2 & B2;
```

Appendix B

Introduction to VHDL

This appendix provides a brief overview of the VHSIC Hardware Description Language briefly referred to as VHDL. This language was developed to be used for documentation, verification, and synthesis of large digital designs.

B.1 Introduction

The VHSIC* Hardware Description Language, henceforth referred to as VHDL is a powerful programming language used to describe hardware designs much the same way we use schematics. It was developed by the Institute of Electrical and Electronics Engineers (IEEE) as Standard VHDL-1076. It was introduced in 1987 as Std 1076-1987, and was upgraded in 1993 as Std 1076-1993. Its popularity stems from the fact that it can be used for documentation, verification, and synthesis of large digital designs.

B.2 The VHDL Design Approach

Consider a combinational digital circuit such as a full adder. Such a circuit is often referred to as a *module*, and it is described in terms of its inputs and outputs. For instance, the full adder module has three inputs and two outputs. In VHDL terminology, the module is referred to as a *design entity*, or simply *design*, and the inputs and outputs are called *ports*. A typical design consists of two or more *blocks*. Thus, the blocks in a full adder are gates and inverters. In other words, blocks are connected together to form a design.

In VHDL it is not necessary to provide a description of the function performed by a block, e.g., a gate or inverter. However, we must describe the function performed by the module, that is, the design. This description is called *functional* or *behavioral* description. Thus, the functional description for the full adder is

$$SUM = X \oplus Y \oplus C_{IN}$$

$$C_{OUT} = XY + XC_{IN} + YC_{IN}$$

In more complex designs the functional description is in the form of an executable program. This will be discussed in a later section.

* *VHSIC is an acronym for Very High Speed Integrated Circuits*

The following simple example illustrates how VHDL is used with a 3-bit counter using three JK flip flops with inputs J_0, K_0, J_1, K_1, J_2, K_2, and outputs Q_0, Q_1, and Q_2. We start the description of the entity (design) by specifying its external interfaces which include a description of its ports (inputs and outputs). The entity declaration in VHDL is as follows:

```
entity count3 is
     generic (prop_delay: Time: = 10 ns);
     port (clock: J0, K0, J1, K1, J2, K2: in bit;
           Q0,Q1,Q2: out bit);
end count3;
```

The first line (**entity**) specifies that the entity `count3` has three inputs and three outputs; these are logic 1 or logic 0 (True or False) values. The second line (**generic**) is optional. If used, it defines the generic constant `prop_delay` which represents the propagation delay. If omitted, the default value 10ns is assumed. The third and fourth lines define the **port** clause, that is, external interfaces (inputs and outputs). The last line (**end**) signifies the end of **entity** `count3`.

The **entity** `count3` is the first part of 3-bit counter design. The second part of the description of the `count3` design is a description of how the design operates. This is defined by the *architecture declaration*. The following is an example of an architecture declaration for the **entity** `count3`.

Figure B.1 shows the structure of `count3` where $T_2 = Q_1Q_2$, $T_1 = Q_2$, and $T_0 = 1$.*

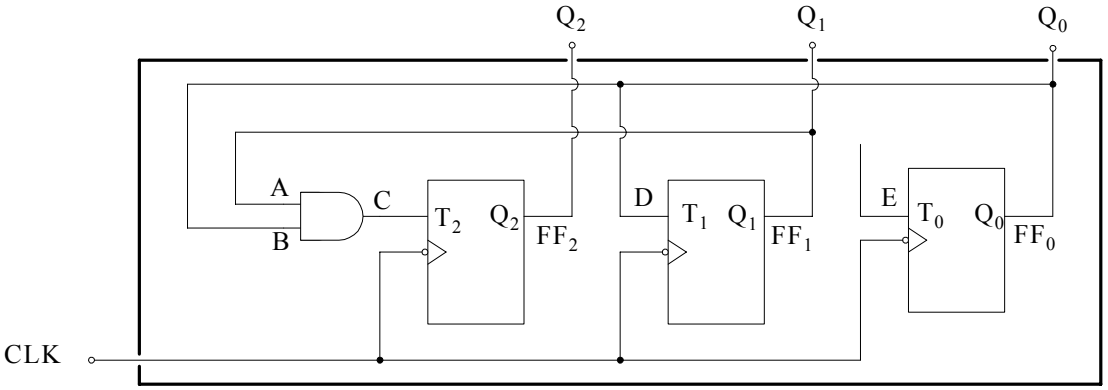


Figure B.1. Structure of `count3`

* *This 3-bit counter is a modification of Example 8.3, Chapter 8, where, for convenience, we have replaced the JK-type flip flops with T-type flip flops and the circuit has been simplified to an up-counter, that is, it counts from 000 to 001 to 010...111 resets to 000 and repeats the count. We have assumed that this circuit is implemented with TTL devices where unconnected inputs behave as logic 1. Accordingly, the input to flip flop* T_0 *is not shown as an external input.*

The description of the **entity** count3 as an architectural body is shown below. However, we must remember that there may be more than one architectural body corresponding to a single entity specification.

```
architecture structure of count3 is
     component T_flip_flop
        port (CLK: in bit; Q: out bit;
     end component;

     component AND_gate
        port (A, B: in bit; C: out bit;
     end component;

     signal C, D, E, FF0, FF1, FF2; bit;

begin
     bit_0: T_flip_flop port map (CLK => Clock,
                                  Q0 => FF0, Q1=>FF1, Q2=>FF2);
     AND: AND_gate port map (A=>FF1, B=>FF0, C=> (A and B));

     bit_1: T_flip_flop port map (CLK => Clock,
                                  Q0 => FF0, Q1=>FF1, Q2=>FF2);

     Q0<=FF0;
     Q1<=FF1;
     Q2<=FF2;

end structure;
```

In the architecture of the above example, two components are declared, T_flip_flop, and AND_gate, and seven internal signals are declared. Each of the components is used to create a component *instance** and the ports (inputs and outputs) of the instances are mapped onto signals and ports of the design (entity). In the above example, bit_0 and bit_1 are instances of the component T_flip_flop. The symbol <= in the last three *signal assignments*, when used in VHDL, specify a relationship among signals, not a transfer of data as in other programming languages. In this case, these three signal assignments update the entity ports whenever the values on the internal signal change.

B.3 VHDL as a Programming Language

This section describes the basic components of the VHDL. We will briefly discuss the element types available in VHDL, the data types and objects, expressions and operators, sequential statements, and subprograms and packages.

* *An component instance, or simply instance, is the actual part such as an inverter, AND gate, flip flop, etc.*

B.3.1 Elements

In VHDL, comments, identifiers, numbers, characters, and strings are components of the category known as *elements*. They are listed below.

B.3.1.1 Comments

Comments in VHDL begin with two hyphens (––) and extend to the end of the line. They are used for explanations and they are ignored by VHDL.

B.3.1.2 Identifiers

Identifiers in VHDL are used as reserved words and as programmer defined names. The syntax is:

```
identifier ::= letter { [ underline ] letter_or_digit }
```

Identifiers are not case sensitive.

B.3.1.3 Literal Numbers

*Literal numbers** may be expressed either in decimal or in a base 2, base 8, or base 16. If the number includes a point, it represents a fractional or mixed number; otherwise it represents an integer.

The syntax for decimal literals is:

```
decimal_number ::= integer [.integer] [exponent]

integer ::= digit { [underline] digit }

exponent ::E [+] integer | E-integer
```

Examples:

```
0      7      142_967_358    235E6    -- integer literals
0.0   0.8     3.14_28     25.34E-7   -- real literals
```

Numbers in bases other than base-10 are defined by:

```
based_number ::= base # based_integer [.based_integer ] # [ exponent]

base ::= integer

based_integer ::= extended_digit { [underline] extended_digit }

extended_digit ::= digit | letter
```

The base and the exponent are expressed in decimal. The exponent indicates the power of the

* *A literal number, as used in a program, is a number that is expressed as itself rather than as a variable's value or the result of an expression. For example, the numbers 12 and 56.15, the character H, the string F5A3, or the Boolean value TRUE are literal numbers.*

base by which the number is multiplied. As we know, the letters A to F are used as extended digits to represent 10 to 15 in hexadecimal numbers.

Examples:

```
2#1101_0101_1010#  8#6532# 16#D5A# --the decimal integer 3418

2#1011.1000_1111_0100# 16#B.8F4#   --the real number 11.5
```

6.12.4 Literal Characters

Literal characters are formed by enclosing an ASCII character in single-quote marks.

Examples:

```
'X'

'*'
```

6.12.5 Literal Strings

Literal strings of characters are enclosed double-quotation marks. To include a double-quote mark itself in a string, a pair of double-quote marks must be put together. A string can be used as a value for an object which is an array of characters.

Examples:

```
""X string""

""""                                    --empty string

""X string in a string:""""X string"".""--contains quotation marks
```

B.3.1.6 Bit Strings

Bit strings provide a convenient way of specifying literal values for arrays of type bit (logic 0s and logic 1s). Arrays are discussed in Subsection B.3.2.5. The syntax is:

```
bit_string_literal ::= base_specifier "" bit_value""

base_specifier ::= B | O | X

bit_value ::= extended_digit ([underline] extended_digit}
```

Base specifier B stands for binary, O for octal, and X for hexadecimal.

Examples:

```
B""1O1O111O""-- length is 8, equivalent to decimal 174
O""145""     -- length is 9, equivalent to B""OO1_10O_101""
X""A7""      -- length is 8, equivalent to B""1010_O111""
```

B.3.2 Data Types

Most of the data types in VHDL are *scalar types*. The scalar types include numbers, physical quantities, and *enumerations** (including enumerations of characters), and there are a number of standard predefined basic types. Various scalar types are combined to form *composite types.* The composite types provided are generally arrays and records. Another class of data types are *pointers* and *files*. A data type can be defined by a type declaration:

```
full_type_declaration ::= type identifier is type_definition

type_definition ::=
   scalar_type_definition
   | composite_type_definition
   | access_type_definition
   | file_type_definition
scalar_type_definition ::=
enumeration_type_definition | integer_type_definition
floating_type_definition | physical_type_definition
composite_type_definition ::=
   array_type_definition
   | record_type_definition
```

Examples will be given in the following sections.

B.3.2.1 Integer Types

An integer type is a range of integer values within a specified range. The syntax is:

```
integer_type_definition : : = range_constraint

range_constraint ::= range range

range : : = simple_expression direction simple_expression

direction ::= to | downto
```

The expressions that specify the range must be integer numbers. Types declared with the keyword **to** are called *ascending ranges*, and those declared with the keyword **downto** are called *descending ranges*. The VHDL standard allows an implementation to restrict the range, but requires that it must at least allow the range –2147483647 to +2 147483647.

Examples:

```
type byte_int is range 0 to 255;

type signed_word_int is range -32768 to 32767;

type bit_index is range 31 downto 0;
```

* *An enumeration type of data is an ordered set of identifiers and characters. These will be discussed in Subsection B.3.2.4.*

There is a predefined integer type called *integer*. The range of this predefined integer includes the range –2147483647 to +2147483647.

B.3.2.2 Physical Types

A *physical type* is a numeric type used to represent physical quantities, such as mass, length, time, voltage, current, power, etc. The declaration of a physical type includes the specification of a base unit but multiple of the base unit, that is, prefixes such as KΩ, μA, etc., are also allowed. The syntax is:

```
physical_type_definition
   range_constraint
      units
         base_unit_declaration
         { secondary_unit_declaration }
      end units
base_unit_declaration ::= identifier;
secondary_unit_declaration : := identifier = physical_literal;
physical_literal ::= abstract_literal I unit_name
```

Examples:

```
type length is range 0 to 1 E9
   units
      um; -- u = micro
      mm = 1000 um;
      cm = 10mm;
      m = 1000 mm;
      in = 25.4 mm;
      ft = 12 in;
      yd = 3 ft;
      rod = 198 in;
      chain = 22 yd;
      furlong = 10 chain;
   end units;
type resistance is range 0 to 1 E8
   units
      ohms;
```

```
        kohms = 1000 ohms;
        Mohms = 1E6 ohms;
    end units;
```

The predefined physical type `time` is used extensively to specify delays in simulations. Its syntax is:

```
type time is range implementation_defined
    units
        fs;  -- f is femto, E-15
        ps = 1000 fs;  -- p is pico, E-12
        ns = 1000 ps;  -- n is nano, E-9
        us = 1000 ns;  -- u is micro, E-6
        ms = 1000 us;  -- m is milli, E-3
        sec = 1000 ms;
        min = 60sec;
        hr = 60min;
    end units;
```

To declare a value of some physical type, we write the number followed by the unit.

Examples:

```
10 um
1 hr
2.2 kohm
100 ns
```

B.3.2.3 Floating Point Types

A floating point type is a discrete approximation to the set of real numbers in a specified range. The precision of the approximation is not defined by the VHDL language standard, but must be at least six decimal digits. The range must include at least -1E38 to +1E38. The syntax is:

```
floating_type_definition := range_constraint
```

Examples:

```
type signal_level is range –15.00 to +15.00;
type probability is range 0.0 to 1.0;
```

There is a predefined floating point type called *real*. The range extends from –1E38 to +1E38.

B.3.2.4 Enumeration Types

As stated earlier, an enumeration type is an ordered set of identifiers or characters. The identifiers and characters within a single enumeration type must be distinct, however they may be reused in several different enumeration types. The syntax is:

```
enumeration_type_definition ::=(enumeration_literal{, enumeration_literal})
enumeration_literal ::= identifier | character_literal
```

Examples:

```
type logic_level is (unknown,low,undriven,high);
type alu_function is (disable,pass,add,subtract,multiply,divide);
type octal_digit is ('0', '1', '2', '3', '4', '5', '6', '7');
```

Other predefined enumeration types are:

```
type severity_level is (note, warning, error, failure);
type boolean is (false, true);
type bit is ('0', '1');
type character is (
      'any ASCII character');
```

Single characters are enclosed in single quotations, e.g., '8', but for control characters, e.g., ESC, no quotations are required.

The characters '0' and '1' are members of both bit and character. Which type is being used, will be determined by the context.

B.3.2.5 Arrays

An *array* is an indexed collection of elements all of the same type. Arrays may be one-dimensional (with just one index) or multidimensional (with two or more indices). An array can be constrained, that is, the bounds for an index are established when the type is defined, or unconstrained, in which the bounds are established subsequently. The syntax is

```
array_type_definition ::=
   unconstrained_array_definition | constrained_array_definition
unconstrained_array_definition ::=
   array (index_subtype_definition {, index_subtype_definition })
       of element_subtype_indication
constrained_array_definition ::=
   array index_constraint of element_subtype_indication
index_subtype_definition :: = type_mark range <>
```

```
index_constraint ::= ( discrete_range {, discrete_range }

discrete_range ::= discrete_subtype_indication | range
```

The subtypes, referred to in the syntax specification above, will be discussed in Subsection B.3.2.7. The symbol <>, referred to as *box*, serves as a place-holder for the index range, which will be filled in later when the array type is used. For instance, an object might be declared to be a vector of 12 elements by giving its type as:

```
vector (1 to 12)
```

In the examples below, the first 4 are constrained array type declarations, and the fifth is an example of an unconstrained type declaration.

Examples:

```
type word is array (15 downto 0) of bit;
type memory is array (address) of word;
type transform is array (1 to 10, 1 to 10) of real;
type register_bank is array (byte range 0 to 264) of integer;
type vector is array (integer range <>) of real;
```

The following two array types are unconstrained predefined array types. The types *positive* and *natural* are subtypes of integer, and are defined in Subsection B.3.2.7 below. The type bit_vector is useful in modeling binary coded representations of values in simulations of digital circuits.

```
type string is array (positive range <>) of character;

type bit_vector is array (natural range <>) of bit;
```

An element of an array object can be referred to by indexing the name of the object. For instance, let us assume that *x* is a one-dimensional array and *y* is a two-dimensional array. Accordingly, the indexed names *x*(1) and *y*(l, 1) refer to elements of these arrays. Also, a one-dimensional array can be referred to by using a range as an index. For example *x*(0 to 31) is a 32-element array which is part of the array *x*.

Quite often, it is convenient to express a literal value in an array type. This can be done using an *array aggregate*, which is a list of element values. For instance, suppose that we wish to write a value of this type containing the elements `'d'`, `'i'`, `'g'`, `'i'`, `'t'`, `'a'`, and `'l'` in that order, and we have an array type available which is declared as:

```
type a is array (1 to 7) of character;
```

and we wish to write a value of this type containing the elements `'d'`, `'i'`, `'g'`, `'i'`, `'t'`, `'a'`, and `'l'` in that order. We can write an aggregate with positional association as follows:

```
('d','i','g','i','t','a','l')
```

Alternatively, we could write an aggregate with the following named association:

```
(1=>'d', 5=>'t', 3=>'g', 7=>'l', 6=>'a', others =>'i')
```

B.3.2.6 Records

Records are collections of some named elements, usually of different types. The syntax is:

```
record_type_definition :: =
    record
       element_declaration
       { element_declaration }
    end record
element_declaration ::= identifier_list : element_subtype_definition;
identifier_list ::= identifier (, identifier)
element_subtype_definition : := subtype_indication
```

Example:

```
type instruction is
record
    op_sys : cpu_op;
    address_mode : mode;
    operand1, operand2: integer range 0 to 31;
end record;
```

Occasionally, we may need to refer to a field of a record object. In this case, we use a selected name. For instance, suppose that r01 is a record object containing a field called f01. Then, the name r01.f01 refers to that field. With arrays, aggregates can be used to write literal values for records. Both positional and named association can be used, and the same rules apply, with record field names being used in place of array index names.

B.3.2.7 Subtypes

The use of a subtype allows the values taken on by an object to be restricted or constrained subset of some base type. The syntax is:

```
subtype_declaration ::= subtype identifier is subtype_indication;
subtype_indication ::= |resolution_function_name|type_mark|constraint|
type_mark ::= type_name | subtype_name
constraint : : = range_constraint | index_constraint
```

There are two classes of subtypes. The first class may constrain values from a scalar type to be within a specified range (a range constraint).

Examples:

```
subtype pin_count is integer range 0 to 200;
subtype digits is character range '0' to '7';
```

The second class may constrain an array type by specifying bounds for the indices.

Examples:

```
subtype id is string(1 to 32);
subtype word is bit_vector(63 downto 0);
```

There are also two predefined numeric subtypes, defined as:

```
subtype natural is integer range 0 to highest_integer
subtype positive is integer range 1 to highest_integer
```

B.3.2.8 Object Declarations

An *object* is a named item which has a value of a specified type. There are three classes of objects: constants, variables, and signals. We will discuss constants and variables in this subsection, and signals in Subsection B.4.2.1.

A *constant** is an object which is assigned a specified value when it is declared, and which should not be subsequently modified. The syntax of a constant is:

```
constant_declaration ::=
constant identifier_list : subtype_indication | := expression |;
```

Examples:

```
constant pi : real := 3.14159;
constant delay : Time := 10 ns;
constant max_size : natural;
```

A *variable* is an object whose value may be changed after it is declared. The syntax is:

```
variable_declaration ::=
   variable identifier_list : subtype_indication | := expression |;
```

The initial value expression, if present, is evaluated and assigned to the variable when it is declared. If the expression is absent, a default value is assigned. The default value for scalar types

* *Some constant declarations, referred to as deferred constants, appear only in package declarations. The initial value must be given in the corresponding package body.We will discuss package declarations in Subsection B.3.5.3.*

is the leftmost value for the type, that is, the first in the list of an enumeration type, the lowest in an ascending range, or the highest in a descending range. If the variable is a composite type, the default value is the composition of the default values for each element, based on the element types.

Examples:

```
variable count: natural := 0;

variable trace : trace_array;
```

Assuming the type `trace_array` is an array of Boolean expressions, the initial value of the variable `trace` is an array with all elements having the value `false`.

We can give an alternate name to a declared object or part of it. This is done using an `alias` declaration. The syntax is:

```
alias_declaration ::= alias identifier : subtype_indication is name ;
```

A reference to an `alias` is interpreted as a reference to the object or part of it corresponding to the `alias`. For instance, the variable declaration

```
variable data_lines: bit_vector(63 downto 0);

alias in_signal : bit_vector(15 downto 0) is instr(63 downto 48);
```

declares the name `in_signal` to be an `alias` for the left-most eight bits of `data_lines`.

B.3.2.9 Attributes

Attributes provide additional information for the types and objects declared in a VHDL description. Some pre-defined attributes are listed below.

An attribute is referenced using the single quotation mark (`‘`) notation. For instance,

```
digital‘attr
```

refers to the attribute `attr` of the type or object `digital`.

Attributes can be used with any scalar type or subtype, with any discrete or physical type or subtype, and with an array type or object. It is customary to denote a type or subtype with the letter T, a member of T with the letter X, an integer with the letter N, and an array type or object with the letter A.

1. For any scalar type or subtype T, the following attributes can be used:

Attribute	Result
T‘left	Left bound of T
T‘right	Right bound of T
T‘low	Lower bound of T
T‘high	Upper bound of T

For an ascending range, `T'left` = `T'low`, and `T'right` = `T'high`.

For a descending range, `T'left` = `T'high`, and `T'right` = `T'low`.

2. For any discrete or physical type or subtype T, X a member of T, and N an integer, the following attributes can be used:

```
Attribute      Result
T'pos(X)       Position number of X in T
T'val(N)       Value at position N in T
T'leftof(X)    Value in T which is one position left from X
T'rightof(X)   Value in T which is one position right from X
T'pred(X)      Value in T which is one position lower than X
T'succ(X)      Value in T which is one position higher than X
```

For an ascending range, `T'leftof(X)` = `T'pred(X)`, and `T'rightof(X)` = `T'succ(X)`.

For a descending range, `T'leftof(X)` = `T'succ(X)`, and `T'rightof(X)` = `T'pred(X)`.

3. For any array type or object A, and N an integer between 1 and the number of dimensions of A, the following attributes can be used:

```
Attribute           Result
A'left(N)           Left bound of index range of dimension N of A
A'right(N)          Right bound of index range of dimension N of A
A'low(N)            Lower bound of index range of dimension N of A
A'high(N)           Upper bound of index range of dimension N of A
A'range(N)          Index range of dimension N of A
A'reverse_range(N)  Reverse of index range of dimension N of A
A'length(N)         Length of index range of dimension N of A
```

B.3.3 Expressions and Operators

An *expression* is a formula combining primaries with operators. Primaries include names of objects, literals, function calls and parenthesized expressions. Operators are listed in Table B-1 in order of decreasing precedence.

The exponentiation (**) operator can have an integer or floating point left operand, but must have an integer right operand. A negative right operand is only allowed if the left operand is a floating point number.

TABLE B.1 Operators and order of precedence in VHDL

Highest Precedence	**	abs	not			
↓	*	/	mod	rem		
↓	+(sign)	-(sign)				
↓	+(addition)	-(subtraction)	&			
↓	=	/=	<	<=	>	>=
Lowest Precedence	and	or	nand	nor	xor	

The absolute value (**abs**) operator works on any numeric type, and **not** implies inversion. The multiplication (*) and division (/) operators work on integer, floating point and physical types. The modulus (**mod**) and remainder (**rem**) operators only work on integer types. The sign operators (+ and –) and the addition (+) and subtraction (–) operators have their conventional meaning on numeric operands. The concatenation operator (&) operates on one-dimensional arrays to form a new array with the contents of the right operand following the contents of the left operand. It can also concatenate a single new element to an array, or two individual elements to form an array. The concatenation operator is most commonly used with strings.

The relational operators =, /=, <, <=, > and >= must have both operands of the same type, and yield Boolean results. The equality and inequality operators (= and /=) can have operands of any type. For composite types, two values are equal if all of their corresponding elements are equal. The remaining operators must have operands which are scalar types or one-dimensional arrays of discrete types.

The logical operators **and**, **or**, **nand**, **nor**, **xor,** and **not** operate on values of type bit or `boolean`, and also on one-dimensional arrays of these types. For array operands, the operation is applied between corresponding elements of each array, yielding an array of the same length as the result. For bit and Boolean operands, **and**, **or**, **nand**, and **nor** they only evaluate their right operand if the left operand does not determine the result. Thus, **and** and **nand** only evaluate the right operand if the left operand is true or `'1'`, and **or** and **nor** only evaluate the right operand if the left operand is false or `'0'`.

B.3.4 Sequential Statements

Sequential statements are those used to modify the state of objects, and to control the flow of execution. They are classified as Variable Assignments, If Statements, Case Statements, Loop Statements, Null Statements, and Assertions. We will discuss these in the following subsections.

B.3.4.1 Variable Assignments

We use an *assignment statement* to assign a value to a variable. The syntax is:

```
variable_assignment_statement ::= target := expression;

target ::= name | aggregate
```

The target of the assignment is an object name, and the value of the expression is given to the named object. The object and the value must have the same base type.

If the target of the assignment is an aggregate, the elements listed must be object names and the value of the expression must be a composite value of the same type as the aggregate. All names in the aggregate are first evaluated, then the expression is evaluated, and finally the components of the expression value are assigned to the named variables. For instance, if a variable `x` is a record with two fields `a` and `b`, then they could be exchanged as

```
(a => x.b, b => x.a) ::= x
```

B.3.4.2 If Statement

We use the *if statement* to choose statements to be executed depending on one or more conditions. The syntax is:

```
if_statement ::=
   if condition then
      sequence_of_statements
   { elsif condition then
      sequence_of_statements }
   |else
      sequence_of_statements |
   end if;
```

The conditions are expressions in Boolean values. These conditions are evaluated successively until one found that yields the value `true`. In that case, the corresponding statement list is executed. Otherwise, if the **`else`** clause is present, its statement list is executed.

B.3.4.3 Case Statement

The case statement allows selection of statements to execute depending on the value of a selection expression. The syntax is:

```
case_statement
   case expression is
      case_statement_alternative
      { case_statement_alternative }
   end case;
case_statement_alternative
   when choices =>
```

```
      sequence_of_statements
choices ::= choice { | choice }
choice ::=
   simple_expression
   | discrete_range
   / element_simple_name
   | others
```

The selection expression must be either a discrete type, or a one-dimensional array of characters. The alternative whose choice list includes the value of the expression is selected and the statement list executed. It is imperative that all choices must be distinct, that is, no value should be duplicated. Moreover, all values must be represented in the choice lists, or the special choice others must be included as the last alternative. If no choice list includes the value of the expression, the others alternative is selected. If the expression results in an array, the choices may be strings or bit strings.

Examples:

```
case element_colour of
   when blue =>
      statements for blue;
   when red | green =>
      statements for red or green;
   when yellow to magenta =>
      statements for these colours;
end case;

case bin_adder of
   when X"000" => perform_add;
   when X"001" => perform_subtract;
   when others => signal_illegal_bin_adder;
end case;
```

B.3.4.4 Loop Statements

Loop statements are similar to loop statement in other programming languages.

```
loop
   do_something;
```

```
end loop;
```

The syntax is:

```
loop_statement ::=
   [loop_label:]
      [iteration_scheme] loop
         sequence_of_statements
      end loop [loop_label];
iteration_scheme ::=
   while condition
   |for loop_parameter_specification
parameter_specification ::=
   identifier in discrete_range
```

If the iteration scheme is omitted, a loop which will repeat the enclosed statements indefinitely will be created.

The while iteration scheme allows a test condition to be evaluated before each iteration. The iteration only proceeds if the test evaluates to true. If the test is false, the loop statement terminates. For instance,

```
while index < length and str(index) /= "loop
index := index + 1;
end loop;
```

The for iteration scheme specifies a number of iterations. The loop parameter specification declares an object that takes on successive values from the given range for each iteration of the loop. Within the statements enclosed in the loop, the object is treated as a constant. The object does not exist after execution of the loop statement. For instance,

```
for item in 1 to last_item loop
   table(item) := 0;
end loop;
```

There are two additional statements which can be used inside a loop to modify the basic pattern of iteration. The 'next' statement terminates execution of the current iteration and starts the subsequent iteration. The 'exit' statement terminates execution of the current iteration and terminates the loop. The syntax is:

```
next_statement ::= next [loop_label] [when condition];
```

```
exit_statement ::= exit [loop_label] [when condition];
```

If the loop label is omitted, the statement applies to the inner-most enclosing loop, otherwise it applies to the named loop. If the `when` clause is present but the condition is false, the iteration continues normally.

Examples:

```
for i in 1 to max_str_len loop
   a(i) := buf(i);
   exit when buf(i) = NUL;
end loop;

outer_loop : loop
   inner_loop: loop
      do_something;
      next outer_loop when temp = 0;
      do_something_else;
   end loop inner_loop;
end loop outer_loop;
```

B.3.4.5 Null Statement

The *null statement* may be used to declare that no action is required in certain cases. It is most often used in `case` statements, where all possible values of the selection expression must be listed as choices, but for some choices no action is required. For instance,

```
case tri_state_device_input is
   when zero => output_zero;
   when one => output_one;
   when high_Z => null;
end case;
```

B.3.4.6 Assertions

An *assertion statement* is used to verify a specified condition and to report if the condition is violated. The syntax is:

```
assertion_statement ::=
   assert condition
      [report expression]
```

```
        [severity expression];
```

If the report clause is present, the result of the expression must be a string. This is a message which will be reported if the condition is `false`. If it is omitted, the default message is "`Assertion violation`". If the severity clause is present, the expression must be of the type `severity_level`. If it is omitted, the default is `error`. A simulator may terminate execution if an assertion violation occurs and the severity value is greater than some implementation dependent threshold. Usually the threshold will be under user control.

B.3.5 Subprograms and Packages

Subprograms are declared in the form of procedures and functions. *Packages* are also used for collecting declarations and objects into modular units. Packages also provide a measure of data abstraction and information hiding.

B.3.5.1 Procedures and Functions

The syntax for *procedure* and *function* subprograms are declared using the syntax:

```
subprogram_declaration ::= subprogram_specification;

subprogram_specification :: =

   procedure designator [(format_parameter_list)]

   | function designator [(format_parameter_list)] return type_mark
```

A subprogram declaration in this form declares the subprogram and specifies the required parameters. The body of statements defining the behavior of the subprogram is deferred. For function subprograms, the declaration also specifies the type of the result returned when the function is called. This form of subprogram declaration is typically used in package specifications as discussed in Subsection B.3.5.3, where the subprogram body is given in the package body, or to define mutually recursive procedures. The syntax is:

```
formal_parameter_list :: = parameter_interface_list

interface_list ::= interface_element {; interface_element}

interface_element ::= interface_declaration

interface_declaration ::=

 interface_constant_declaration

 |interface_signal_declaration

 |interface_variable_declaration

interface_constant_declaration ::=

 [constant] identifier_list:[in] subtype_indication [:= static_expression]

interface_variable_declaration ::=
```

```
[variable] identifier_list:[mode] subtype_indication [:= static_expression]
```

A procedure with no parameters is declared as,

```
procedure reset;
```

This declaration defines `reset` as a procedure with no parameters; the statement body will be given subsequently. A procedure call to `reset` would be:

```
reset;
```

The following is a declaration of a procedure with some parameters:

```
procedure increment_reg(variable reg: inout word_64;
   constant incr: in integer := 1);
```

In the declaration above, the procedure `increment_reg` has two parameters, the first is `reg` and the second is `incr`. The first parameter, i.e., `reg` is a variable parameter, and will be treated as a variable object. Accordingly, when the procedure is called, the actual parameter associated with `reg` must be a variable. The mode of `reg` is `inout`, which means that `reg` can be both read and assigned to. Other possible modes for subprogram parameters are `in`, which means that the parameter may only be read, and `out`, which means that the parameter may only be assigned to. If the mode is `inout` or `out`, then the word variable is assumed and thus it can be omitted.

The second parameter, `incr`, is a constant parameter and it is treated as a constant object in the subprogram statement body. The actual parameter associated with `incr` when the procedure is called must be an expression. With the mode of the parameter, `in`, the word `constant` could be omitted. The expression after the assignment operator is a default expression and it is used if no actual parameter is associated with `incr`.

A call to a subprogram includes a list of actual parameters to be associated with the formal parameters. This association list can be position, named, or a combination of both. A call with positional association lists the actual parameters in the same order as the formals. For instance,

```
increment_reg(index_reg, offset-3); -- add value to index_reg
increment_reg(prog_counter);  -- add 1 (default) to prog_counter
```

A call with named association explicitly gives the formal parameter name to be associated with each actual parameter, so the parameters can be in any order. For instance,

```
increment_reg(incr => offset-3, reg => index_reg);
increment_reg(reg => prog_counter);
```

We observe that the second line in both declarations above do not specify a value for the formal parameter `incr`, so the default value ,1, is used.

The syntax for a function subprogram declaration is:

```
function byte_to_int(byte : word_8) return integer;
```

The function has one parameter. For functions, the parameter mode must be **in**, and this is assumed if not explicitly specified. If the parameter class is not specified, it is assumed to be constant. The value returned by the body of this function must be an integer.

When the body of a subprogram is specified, the syntax used is:

```
subprogram_body ::=
   subprogram_specification is
      subprogram_declarative_part
   begin
      subprogram_statement_part
   end [designator]
subprogram_declarative_part :: = { subprograni_declarative_item }
subprogram_statement_part ::= { sequential_statement )
subprogram_declarative_item :: =
   subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| alias_declaration
```

The declarative items listed after the subprogram specification declare items to be used within the subprogram body. The names of these items are not visible outside of the subprogram, but are visible inside the declared subprograms. Furthermore, these items hide any items with the same names declared outside the subprogram.

When the subprogram is called, the statements in the body are executed until either the end of the statement list is encountered, or a return statementis executed. The syntax of a `return` statement is:

```
return_statement ::= return [ expression]
```

If a return statement occurs in a procedure body, it must not include an expression. There must be at least one return statement in a function body, it must have an expression, and the function must be completed by executing a return statement. The value of the expression is the valued returned to the function call.

When using function subprograms, no visible variable declared outside the function body should be assigned to or altered by the function. This includes passing a non-local variable to a procedure as a variable parameter with mode `out` or `inout`. The important result of this rule is that functions can be called without them having any effect on the environment of the call. For instance,

```
function byte_to_int(byte : word_16) return integer is
   variable result: integer := 0;
begin
   for index in 0 to 15 loop
      result result*2 + bit'pos(byte(index));
   end loop;
   return result;
   end byte_to_int;
```

B.3.5.2 Overloading

Overloading refers to the condition where two subprograms have the same name, but the number or base types of parameters is different. When a subprogram call is made using an overloaded name, the number of actual parameters, their order, their base types, and the corresponding formal parameter names (if named association is used) are used to determine which subprogram is meant. If the call is a function call, the result type is also used. For example, suppose we declared the two subprograms:

```
function check_limit(value : integer) return boolean;
function check_limit(value : word_16) return boolean;
```

Which of the two functions is called depends on whether a value of type `integer` or `word_16` is used as the actual parameter. For instance,

```
test := check_limit(4095)
```

would call the first function above, and

```
test := check_limit(X"BF5A")
```

would call the second function.

The designator used to define a subprogram can be either an identifier or a string representing any of the operator symbols listed in Subsection B.3.3, Table B.1. The latter case allows extra operand types to be defined for those operators. For example, the addition operator might be overloaded to add `word_32` operands by declaring a function:

```
function "+" (a, b : word_32) return word_32 is
begin
   return int_to_word_32(word_32_to_int(a) + word_32_to_int(b));
```

```
end "+";
```

Within the body of this function, the addition operator is used to add integers, since its operands are both integers. However, in the expression:

```
X"1000_0010" + X"0000_FFD0"
```

the operands to the addition operator are both of type `word_32`, and thus a newly declared function is called. It is also possible to call operators using the prefix notation used for ordinary subprogram calls, for example:

```
"+" (X"1000_0010",X"0000_FFD0")
```

B.3.5.3 Package and Package Body Declarations

A *package* is a collection of types, constants, subprograms and other items, usually intended to implement some particular service or to isolate a group of related items. In particular, the details of constant values and subprogram bodies can be hidden from users of a package, with only their interfaces made visible.

A package may be split into two parts: a package declaration, which defines its interface, and a package body, which defines the deferred details. The body part may be omitted if there are no deferred details. The syntax is:

```
package_declaration ::=
   package identifier is
      package_declarative_part
   end [ package_simple_name ];
package_declarative_part ::= { package_declarative_item }
package_declarative_item ::=
   subprogram_declaration
   | type_declaration
   | subtype_declaration
   | constant_declaration
   | alias_declaration
   | use_clause
```

The declarations define items that are to be visible to users of the package, and are also visible inside the package body. For instance,

```
package data_types is
   subtype address is bit_vector(24 downto 0);
```

```
    subtype data is bit_vector(15 downto 0);
    constant vector_table_loc : address;
    function data_to_int(value : data) return integer;
    function int_to_data(value : integer) return data;
end data_types;
```

In the example above, the value of the constant vector_table_loc and the bodies of the two functions are deferred, so a package body needs to be given. The syntax for a package body is:

```
package_body ::=
    package body package_simple_name is
        package_body_declarative_part
    end [package_simple_name];
package_body_declarative_part ::= { package_body_declarative_item }
package_body_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | alias_declaration
    | use_clause
```

Subprogram bodies may be included in a package body, whereas only subprogram interface declarations may be included in the package interface declaration. The body for the package data_types shown above might be written as:

```
package body data_types is
    constant vector_table_bc: address := X"FFFF00";
    function data_to_int(value : data) return integer is
        body of data_to_int
    end data_to_int;
    function int_to_data(value : integer) return data is
        body of int_to_data
    end int_to_data;
end data_types;
```

In this package body, the value for the constant is specified, and the function bodies are given. The subtype declarations are not repeated, as those in the package declarations are visible in the package body.

B.3.5.4 Package Use and Name Visibility

Once a package has been declared, items declared within it can be used by prefixing their names with the package name. For example, given the package declaration in Section2.4.3 above, the items declared might be used as follows:

```
variable PC : data_types.address;
int_vector_loc := data_types.vector_table_loc + 4*int_level;
offset := data_types.data_to_int(offset_reg);
```

Quite often, it is convenient to refer to names from a package without having to qualify each use with the package name. This is be done using a `use` clause in a declaration region. The syntax is:

```
use_clause ::= use selected_name { , selected_name }
selected_name ::= prefix suffix
```

The effect of the `use` clause is that all of the listed names can subsequently be used without having to prefix them. If all of the declared names in a package are to be used in this way, we can use the special suffix all, for example:

```
use data_types.all;
```

B.4 Structural Description

In our previous sections, we introduced some terminology for describing the structure of a digital system. We discussed lexical elements, data types and objects, expressions and operators sequential statements, and subprograms and packages. We also introduced some terminology for describing the structure of a digital system. In this section, we will examine in more detail how the structure is described in VHDL.

B.4.1 Entity Declarations

We saw earlier that a typical digital system is usually designed as a hierarchical collection of modles. Each module has a set of ports which constitute its interface to the outside world. In VHDL, an entity is such a module which may be used as a component in a design, or which may be the top level module of the design. The syntax for declaring an entity is as follows:

```
entity_declaration ::=
   entity identifier is
      entity_header
      entity_declarative_part
```

```
    [begin
       entity_statement_part]
    end [entity_simple_name]
entity_header ::=
    [formal_generic_clause]
    [formal_port_clause]
generic_clause ::= generic (generic_list);
generic_list ::= generic_interface_list
port_clause ::= port (port_list);
port_list ::= port_interface_list
entity_declarative_part ::= { entity_declarative_item }
```

The entity declarative part may be used to declare items which are to be used in the implementation of the entity. Usually such declarations will be included in the implementation itself, so they are only mentioned here for completeness. Also, the optional statements in the entity declaration may be used to define some special behavior for monitoring operation of the entity.

The entity header is the most important part of the entity declaration. It may include specification of generic constants, which can be used to control the structure and behavior of the entity, and ports, which channel information into and out of the entity.

The generic constants are specified using an interface list similar to that of a subprogram declaration. All of the items must be of class constant. As a reminder, the syntax of an interface constant declaration is:

```
interface_constant_declaration ::=
    [constant] identifier_list:[in] subtype_indication
                                        [:=static_expression]
```

The actual value for each generic constant is passed in when the entity is used as a component in a design.

The entity ports are also specified using an interface list, but the items in the list must all be of class signal. This is a new kind of interface item not previously discussed. The syntax is:

```
interface_signal_declaration ::=
    [signal] identifier_list : [mode] subtype_indication [bus]
       [:=static_expression]
```

Since the class must be signal, the word **signal** is assumed and thus it can be omitted. The word **bus** may be used if the port is to be connected to more than one output.

As with generic constants, the actual signals to be connected to the ports are specified when the entity is used as a component in a design. Let us consider the following entity declaration:

```
entity cpu is
   generic (max_clock_freq : frequency := 30 MHz);
   port (clock: in bit;
         address : out integer;
         data: inout word_32;
         control : out proc_control;
         ready: in bit);
end processor;
```

Here, the generic constant `max_clock_freq` is used to specify the timing behavior of the entity. The code describing the entity's behavior would use this value to determine delays in changing signal values.

Next, let us consider the following entity declaration showing how generic parameters can be used to specify a class of entities with varying structure:

```
entity ROM is
   generic (width, depth : positive);
   port (enable : in bit;
         address : in bit_vector(depth-1 downto 0);
         data : out bit_vector(width-1 downto 0));
end ROM;
```

Here, the two generic constants are used to specify the number of data bits and address bits respectively for the ROM. We observe that no default value is given for either of these constants. When the entity is used as a component, actual values must be supplied for them.

Let us also consider the following entity declaration with no generic constants or ports:

```
entity test_bench is
end test_bench;
```

We observe that this entity declaration has no ports, that is, it has no external connections. This entity declaration could be used loop feedback circuit such as that shown in Figure B.2. A top-level entity for a *Design Under Test* (*DUT*) is used as a component in a loop feedback circuit circuit with another entity denoted as *Test Generator* (TG) whose purpose is to generate test values. The values on signals can be traced using a simulation monitor, or checked directly by the test generator.

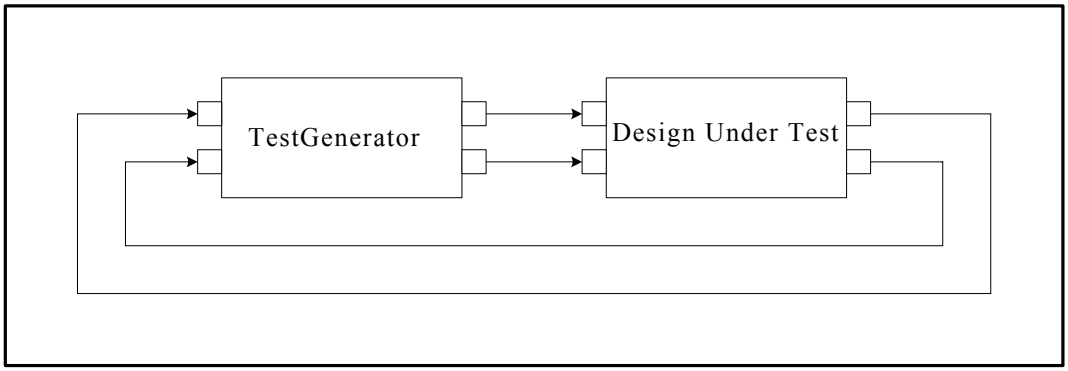


Figure B.2. Loop feedback circuit

B.4.2 Architecture Declarations

An *architectural declaration* follows the entity declaration. We use an architectural declaration to provide one or more implementations of the entity in the form of *architecture bodies*. Each architecture body can describe a different view of the entity. An architecture body is declared using the syntax:

```
architecture_body :: =
   architecture identifier of entity_name is
      architecture_declarative_part
   begin
      architecture_statement_part
   end [architecture_simple_name]
architecture_declarative_part : = { block_declarative_item }
architecture_statement_part = { concurrent_statement }
block_declarative_item ::=
   subprogram_declaration
   | subprogram_body
   | type_declaration
   | subtype_declaration
   | constant_declaration
   | signal_declaration
   | alias_declaration
   | component_declaration
   | configuration_specification
```

```
   | use_clause
concurrent_statement ::=
   block_statement
   | component_instantiation_statement
```

The declarations in the architecture body define items that will be used to construct the design description. Signals and components should be declared in the architecture body to be used to construct a structural description in terms of component instances as described in the subsections below.

B.4.2.1 Signal Declarations

Signals are used to connect submodules in a design. The syntax is:

```
signal_declaration ::=
signal identifier_list : subtype_indication [signal_kind] [:=expression];
signal_kind ::= register | bus
```

If we omit the signal, the subtype specified will be ordinary signal. The expression in the declaration is used to give the signal an initial value during the initialization phase of simulation. If the expression is omitted, a default initial value will be assigned.

Ports of an object are treated exactly as signals within that object.

B.4.2.2 Blocks

A *block* is a unit of module structure, with its own interface, connected to other blocks or ports by signals. The submodules in an architecture body are described as blocks. The syntax is:

```
block_statement ::=
   block_label:
      block [(block_header)]
         block_declarative_part
      begin
         block_statement_part
      end block [block_label];
block_header ::=
   [generic_clause
   [generic_map_aspect]]
   [port_clause
   [port_map_aspect;]]
```

```
generic_map_aspect ::= generic map (generic_association_list)
port_map_aspect : := port map (port_association_list)
block_declarative_part ::= { block_declarative_item }
block_statement_part ::= { concurrent_statement }
```

The block header defines the interface to the block in much the same way as an entity header defines the interface to an entity. The generic association list specifies values for the generic constants, evaluated in the context of the enclosing block or architecture body. The port map association list specifies which actual signals or ports from the enclosing block or architecture body are connected to the block's ports. Note that a block statement part may also contain block statements, so a design can be composed of a hierarchy of blocks, with behavioral descriptions at the bottom level of the hierarchy.

As an illustration, let us declare a structural architecture of the cpu entity that we discussed earlier. If we separate the cpu into a control unit and a data path section, we can write a description as a pair of interconnected blocks as shown below.

```
architecture block_structure of cpu is
   type data_path_control is...;
   signal internal_control : data_path_control;
begin
   control_unit : block
      port (clk:in bit;
         bus_control : out proc_control;
         bus_ready: in bit;
         control : out data_path_control);
      port map (clk => clock,
         bus_control => control, bus_ready => ready;
         control => internal_control);
      declarations for control_unit
   begin
      statements for control_unit
   end block control_unit;
   data_path : block
      port (address: out integer;
         data: inout word_32;
```

```
            control : in data_path_control);
        port map (address => address, data => data,
            control => internal_control);
        declarations for data_path
    begin
        statements for data_path
    end block data_path;
end block_structure;
```

The control unit block has ports `clk`, `bus_control` and `bus_ready`, which are connected to the `cpu` entity ports. It also has an output port for controlling the data path, which is connected to a signal declared in the architecture. That signal is also connected to a control port on the data path block. The address and data ports of the data path block are connected to the corresponding entity ports. The advantage of this modular decomposition is that each of the blocks can then be developed independently, with the only effects on other blocks being well defined through their interfaces.

B.4.2.3 Component Declarations

An architecture body can also make use of other entities described separately and placed in design libraries. In order to do this, the architecture must declare a component, which can be thought of as a template defining a virtual design entity, to be instantiated within the architecture. The syntax is:

```
component_declaration ::=
    component identifier
        [local_generic_clause]
    end component;
```

Examples:

```
component xor
    generic (Tpd : Time := 1 ns);
    port (A, B: in logic_level;
        C: out logic_level);
end component;
```

This example declares a `xor` gate with a generic parameter specifying its propagation delay. Different instances can later be used with possibly different propagation delays.

```
component rom
   generic (data_bits, addr_bits : positive);
   port (en : in bit;
      addr: in bit_vector(depth-1 downto 0);
      data : out bit_vector(width-1 downto 0));
end component;
```

This example declares a `rom` component with address depth and data width dependent on generic constants. This component can be used as a template for the `rom` entity.

B.4.2.4 Component Instantiation

A component defined in an architecture may be instantiated using the syntax:

```
component_instantiation_statement ::=
   instantiation_label
      component_name
      [generic_map_aspect]
      [port_map_aspect];
```

This syntax that the architecture contains an instance of the named component, with actual values specified for generic constants, and with the component ports connected to actual signals or entity ports.

The example component declared in the previous section with ROM can be instantiated as:

```
parameter_rom: read_only_memory
   generic map (data_bits => 16, addr_bits => 8);
   port map (en => rom_sel, data => param, addr => a(7 downto 0);
```

In the instance above, values are specified for the address and data port sizes. We observe that the actual signal associated with the port `addr` is a part of an array signal. This illustrates that a port which is an array can be connected to part of a signal which is a larger array, and this is a very common practice with bus signals.

B.5 Behavioral Description

It was stated earlier that the behavior of a digital system could be described in terms of programming language notation. In this section we describe how these are extended to include statements for modifying values on signals, and means of responding to the changing signal values.

B.5.1 Signal Assignment

A signal assignment schedules one or more transactions to a signal or port. The syntax is:

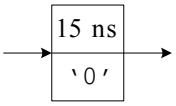
```
signal_assignment_statement ::= target <= [transport] waveform;
target ::= name |aggregate
waveform ::= waveform_element {, waveform_ element}
waveform_element ::=
   value_expression [after time_expression]
   |null [after time_expression]
```

The target must represent a signal, or be an aggregate of signals. If the time expression for the delay is omitted, it defaults to 0 fs. This means that the transaction will be scheduled for the same time as the assignment is executed, but during the next simulation cycle.

Each signal has associated with it a list of transactions giving future values for the signal, and this list is referred to as *projected output waveform*. A signal assignment adds transactions to this waveform. For instance, the signal assignment:

```
s <= '0' after 10 ns;
```

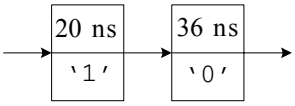
will cause the signal enable to assume the value true 10 ns after the assignment is executed. It is customary to represent the projected output waveform graphically by showing the transactions along a time axis. Thus, if the above assignment were executed at time 5 ns, the projected waveform would shown as:



and when simulation time reaches 15 ns, this transaction will be processed and the signal updated. As another example, suppose that at time 16 ns, the following assignment is executed:

```
s <= '1' after 4 ns, '0' after 20 ns;
```

The two new transactions are added to the projected output waveform:



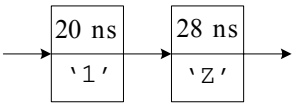
We must remember that when multiple transactions are listed in a signal assignment, the delay times specified must be in ascending order.

If a signal assignment is executed, and there are already old transactions from a previous assignment on the projected output waveform, some of the old transactions may be deleted. The way this is done depends on whether the word transport is included in the new assignment. if it is included, the assignment is said to use *transport delay*. In this case, all old transactions scheduled

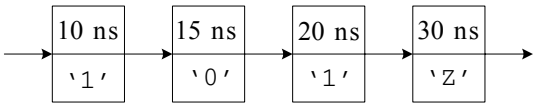
to occur after the first new transaction are deleted before the new transactions are added. It is as though the new transactions supersede the old ones. For instance, if the projected output waveform is as shown above, and afterwards the assignment:

```
s <= transport 'Z' after 10 ns;
```

were executed at time 18 ns, then the transaction scheduled for 36 ns would be deleted, and the projected output waveform would become:



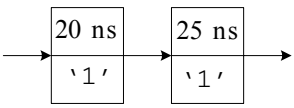
Another kind of delay, referred to as *inertial delay*, is used to model devices that do not respond to input pulses shorter than their output delay. An inertial delay is specified by omitting the word transport from the signal assignment. When an inertial delay transaction is added to a projected output waveform, all old transactions scheduled to occur after the new transaction are deleted, and the new transaction is added, as in the case of transport delay. Afterwards, all old transactions scheduled to occur before the new transaction are examined. If there are any with a different value from the new transaction, then all transactions up to the last one with a different value are deleted. The remaining transactions with the same value are left. For instance, suppose that the projected output waveform at time 0 ns is as shown below,



and the assignment

```
s <= 1 after 25 ns;
```

is executed afterwards at 0 ns. The new projected output waveform will be:



When a signal assignment with multiple waveform elements is specified with inertial delay, only the first transaction uses inertial delay; the rest are treated as being transport delay transactions.

B.5.2 Process and the Wait Statement

The primary unit of behavioral description in VHDL is the process. A *process* is a sequential body of code which can be activated in response to changes in state. When more than one process is activated at the same time, they execute concurrently. A process is specified in a process statement, with the syntax:

```
process_statement ::=
```

```
    [ process_label:]
       process [(sensitivity_list)]
          process_declarative_part
       begin
          process_statement_part
       end process [process_label];
process_declarative_part : = {process_declarative_item}
process_declarative_item ::=
   subprogram_declaration
   | subprogram body
   | type_declaration
   | subtype_declaration
   | constant declaration
   | variable declaration
   | alias_declaration
   | use_clause
process_statement_part ::= { sequential_statement}
sequential_statement ::=
   wait_stament
   | assertion_statement
   | signal_assignment_statement
   | variable_assignment_statement
   | procedure_call_statement
   | if_statement
   | case_statement
   | loop_statement
   | next_statement
   | exit_statement
   | return_statement
   | null statement
```

A process statement is a concurrent statement that can be used in an architecture body or block. The declarations define items that can be used locally within the process. The variables may be defined here and used to store state in a model.

A process may contain a number of signal assignment statements for a given signal, which together form a *driver for the signal*. Normally, there may only be one driver for a signal, and thus the code that determines the value of a signal is confined to one process.

A process is activated initially during the initialization phase of simulation. It executes all of the sequential statements, and then repeats starting again with the first statement. A process may suspended itself by executing a wait statement. This is of the form:

```
wait statement ::=
   wait [sensitivity_clause] [condition_clause] [timeout_clause];
sensitivity_clause :: = on sensitivity_list
sensitivity_list ::= signal_name {, signal_name}
condition_clause ::= until condition
timeout_clause for time_expression
```

The sensitivity list of the wait statement specifies a set of signals to which the process is sensitive while it is suspended. When an event occurs on any of these signals, that is, the value of the signal changes, the process resumes and evaluates the condition. If it is true or if the condition is omitted, execution proceeds with the next statement, otherwise the process re-suspends. If the sensitivity clause is omitted, the process is sensitive to all of the signals mentioned in the condition expression. The timeout expression must evaluate to a positive duration, and indicates the maximum time for which the process will wait. If it is omitted, the process may wait indefinitely.

If a sensitivity list is included in the header of a process statement, then the process is assumed to have an implicit wait statement at the end of its statement part. The sensitivity list of this implicit wait statement is the same as that in the process header. In this case the process may not contain any explicit wait statements. Below is an example of a process statements with a sensitivity list.

```
process (reset, clock)
   variable state : bit := false:
begin
   if reset then
      state := false;
   elsif clock = true then
      state := not state;
   end if;
   q <= state after prop_delay;
```

```
    --implicit wait on reset, clock
end process;
```

During the initialization phase of simulation, the process is activated and assigns the initial value of state to the signal q. It then suspends at the implicit wait statement indicated in the comment. When either reset or clock change value, the process is resumed, and execution repeats from the beginning.

The example below describes the behavior of a synchronization device called a *Muller-C element* used to construct asynchronous logic. The output of the device starts at the value 0, and stays at this value until both inputs are 1í, at which time the output changes to 1. The output stays 1 until both inputs are 0, at which time the output changes back to 0.

```
mulleLc_2 : process
begin
   wait until a = '1' and b = '1';
   q <= '1';
   wait until a = '0' and b = '0';
   q <= '0';
end process muller_c_2;
```

This process does not include a sensitivity list, so explicit wait statements are used to control the suspension and activation of the process. In both wait statements, the sensitivity list is the set of signals a and b, determined from the condition expression.

B.5.3 Concurrent Signal Assignment Statements

Often, a process describing a driver for a signal contains only one signal assignment statement. VHDL provides a convenient short-hand notation, called a *concurrent signal assignment* statement, for expressing such processes. The syntax is:

```
concurrent_signal_assignment_statement ::=
   [label :] conditional_signal_assignment
   [label :] selected_signal_assignment
```

For each kind of concurrent signal assignment, there is a corresponding process statement with the same meaning.

B.5.3.1 Conditional Signal Assignment

A *conditional signal assignment* statement is a shorthand for a process containing signal assignments in an if statement. The syntax is:

```
conditional_signal_assignment ::= target <= options conditional_waveforms
```

```
options [guarded] [transport]
conditional_waveforms
{ waveform when condition else }
waveform
```

Use of the word **guarded** is not discussed here. If the word **transport** is included, then the signal assignments in the equivalent process use transport delay.

Suppose we have the following conditional signal assignment:

```
s <= waveform_1 when condition_1 else
   waveform_2 when condition_2 else
   ....
   waveform_n;
```

The equivalent process is:

```
process
   if condition_1 then
      s <= waveform_1;
   elsif condition_2 then
      s <= waveform_2;
   elsif ....
   else
      s <= waveform_n;
   wait [sensitivity_clause];
end process;
```

If none of the waveform value expressions or conditions contains a reference to a signal, the wait statement at the end of the equivalent process has no sensitivity clause. This means that after the assignment is made, the process suspends indefinitely. For example, the conditional assignment:

```
reset <= '1', '0' after 10 ns when short_pulse_required else
         '1', '0' after 50 ns;
```

schedules two transactions on the signal `reset`, then suspends for the rest of the simulation.

If there are references to signals in the waveform value expressions or conditions, then the `wait` statement has a sensitivity list consisting of all of the signals referenced. Thus the conditional assignment:

```
mux_out <= 'Z' after Tpd when en = '0' else
```

```
            in_0 after Tpd when sel = '0' else
            in_1 after Tpd;
```

is sensitive to the signals en and sel. The process is activated during the initialization phase, and thereafter whenever either of en or sel changes

The degenerate case of a conditional signal assignment, containing no conditional parts, is equivalent to a process containing just a signal assignment statement. Thus,

```
s <= waveform;
```

is equivalent to:

```
process
   s <= waveform;
   wait [sensitivity_clause];
end process;
```

B.5.3.2 Selected Signal Assignment

A *selected signal assignment* statement is a shorthand for a process containing signal assignments in a case statement. The syntax is:

```
selected_signal_assignment ::=
   with expression select
      target <= options selected_waveforms;
selected_waveforms
   {waveform when choices,}
   waveform when choices
choices ::= choice {| choice}
```

The options part is the same as for a conditional signal assignment.

Suppose we have a selected signal assignment:

```
with expression select
   s <= waveform_1 when choice_list_1,
        waveform_2 when choice_list_2,
        ....
        waveform_n when choice_list_n;
```

Then the equivalent process is:

```
process
   case expression is
      when choice_list_1 =>
         s <= waveform_1;
      when choice_list_2=>
         s <= waveform_2;
         ....
      when choice_list_n=>
         s <= waveform_n;
end case;
   wait [sensitivity_clause]
end process;
```

The sensitivity list for the wait statement is determined in the same way as for a conditional signal assignment. That is, if no signals are referenced in the selected signal assignment expression or waveforms, the wait statement has no sensitivity clause. Otherwise, the sensitivity clause contains all the signals referenced in the expression and waveforms.

An example of a selected signal assignment statement:

```
with alu_function select
   alu_result <= op1 + op2 when alu_add | alu_incr,
                 op1 - op2 when alu_subtract,
                 op1 and op2 when alu_and,
                 op1 or op2 when alu_or,
                 op1 and not op2 when alu_mask;
```

In this example, the value of the signal `alu_function` is used to select which signal assignment to `alu_result` to execute. The statement is sensitive to the signals `alu_function`, `op1` and `op2`, and thus whenever any of these change value, the selected signal assignment is resumed.

B.6 Organization

In the previous sections we described the various facilities of VHDL. In this section we will illustrate how they are all combined together to form a complete VHDL description of a digital system.

B.6.1 Design Units and Libraries

When we write VHDL descriptions, we write them in a *design file*, then we invoke a compiler to analyze them and insert them into a *design library*. A number of VHDL constructs may be separately analyzed for inclusion in a design library. These constructs are called *library units*. The *primary library units* are entity declarations, package declarations and configuration declarations as discussed in the next subsection. The *secondary library units* are architecture bodies and package bodies. These library units depend on the specification of their interface in a corresponding primary library unit, so the primary unit must be analyzed before any corresponding secondary unit.

A design file may contain a number of library units. The structure of a design file can be specified by the syntax:

```
design_file design_unit { design_unit }

design_unit : = context_clause library_unit

context_clause ::= { context_item }

context_item ::= library_clause | use_clause

library_clause ::= library logical_name_list;

logical_name_list ::= logical_name {, logical_name }

library_unit ::= primary_unit | secondary_unit

primary_unit ::=
  entity_declaration | configuration_declaration | package_declaration

secondary_unit ::= architecture_body | package_body
```

Libraries are referred to using identifiers called *logical names*. This name must be translated by the host operating system into an implementation dependent storage name. For instance, design libraries may be implemented as database files, and the logical name might be used to determine the database file name. Library units in a given library can be referred to by prefixing their name with the library logical name. For example, `ttl_lib.ttl_10` would refer to the unit `ttl_10` in library `ttl_lib`.

The context clause preceding each library unit specifies which other libraries it references and which packages it uses. The scope of the names made visible by the context clause extends until the end of the design unit.

There are two special libraries which are implicitly available to all design units, and thus do not need to be named in a library clause. The first of these is called `work`, and refers to the working design library into which the current design units will be placed by the analyzer. Hence in a design unit, the previously analyzed design units in a design file can be referred to using the library name `work`.

The second special library is called `std`, and contains the package *standard*. The package standard contains all of the predefined types and functions. All of the items in this package are implicitly visible, and thus no use clause is necessary to access them.

B.6.2 Configurations

In Sections B.4.2.3 and B.4.2.4 we discussed how a structural description can declare a component specification and create instances of components. We mentioned that a component declared can be thought of as a template for a design entity. The binding of an entity to this template is achieved through a configuration declaration. This declaration can also be used to specify actual generic constants for components and blocks. Accordingly, the configuration declaration plays a pivotal role in organizing a design description in preparation for simulation or other processing.

The syntax of a configuration declaration is:

```
configuration_declaration ::=
    configuration identifier of entity_name is
       configuration_declarative_part
       block_configuration
    end [configuration_simple_name];
configuration_declarative_part ::= {configuration_declarative_item}
configuration_declarative_item ::= use_clause
block_configuration ::=
    for block_specification
       {use_clause}
       {configuration_item}
    end for;
block_specification :: = architecture_name | block_statement_label
configuration_item :: = block_configuration | component_configuration
component_configuration ::=
    for component_specification
       [use binding_indication;]
       [block_configuration]
    end for;
component_specification ::= instantiation_list : component_name
instantiation_list ::=
```

```
    instantiation_label {, instantiation_label}
    | others
    | all
binding_indication ::=
    entity_aspect
    [generic_map_aspect]
    [port_map_aspect]
entity_aspect ::=
    entity entity_name [ (architecture_identifier) ]
    | configuration configuration_name
    | open
generic_map_aspect ::= generic map (generic_association_list)
port_map_aspect ::= port map (port_association_list)
```

The declarative part of the configuration declaration allows the configuration to use items from libraries and packages. The outermost block configuration in the configuration declaration defines the configuration for an architecture of the named entity. An example of an entity and architectural body is given below.

```
entity cpu is
   generic (max_clock_speed: frequency := 1000 MHz);
   port (port list);
end cpu;
architecture block_structure of cpu is
   declarations
begin
   control_unit: block
      port (port list);
      port map (association list);
      declarations for control_unit
   begin
      statements for control_unit
   end block control_unit;
```

```
   data_path : block
      port (port list);
      port map (association list);
      declarations for data_path
   begin
      statements for data_path
   end block data_path;
end block_structure
```

The overall structure of a configuration declaration for this architecture is listed below.

```
configuration test_config of cpu is
   use work.cpu_types.all
   for block_structure
      configuration items
   end for;
end test_config;
```

In the configuration declaration above, the contents of a package called `cpu_types` in the current working library are made visible, and the block configuration refers to the architecture `block_structure` of the entity processor.

Within the block configuration for the architecture, the submodules of the architecture may be configured. These submodules include blocks and component instances. A block is configured with a nested block configuration. For instance, the blocks in the above architecture can be configured as shown below.

```
configuration test_config of cpu is
   use work.processor_types.all
   for block_structure
      for control_unit
         configuration items
      end for;
      for data_path
         configuration items
      end for;
   end for;
```

```
end test_config;
```

Where a submodule is an instance of a component, a component configuration is used to bind an entity to the component instance. To illustrate, suppose the `data_path` block in the above example contained an instance of the component `alu`, declared as shown below.

```
data_path : block
   port (port list);
   port map (association list);
   component alu
      port (function : in alu_function;
         op1, op2 : in bit_vector_32;
         result: out bit_vector_32);
   end component;
   other declarations for data_path
begin
   data_alu : alu
      port map (function => alu_fn, op1 => b1, op2 => b2,
                                          result => alu_r);
   other statements for data_path
end block data_path;
```

Now, let us suppose also that a library `project_cells` contains an entity called `alu_cell` with an architecture called `behavior`, and it is defined as shown below.

```
entity alu_cell is
   generic (width : positive);
   port (function_code : in alu_function;
         operand1, operand2 : in bit_vector(width-1 downto 0);
         result: out bit_vector(width-1 downto 0);
         flags : out alu_f lags);
end alu_cell;
```

The entity above matches the `alu` component template, since its operand and result ports can be constrained to match those of the component, and the flags port can be left unconnected. A block configuration for `data_path` could be specified as shown below.

```
for data_path
   for data_alu : alu
      use entity project_cells.alu_cell(behavior)
         generic map (width => 32)
         port map (function_code => function, operand1 => op1,
                   operand2 => op2, result => result, flags => open);
   end for;
   other configuration items
end for;
```

Alternatively, if the library also contained a configuration called `alu_struct` for an architecture structure of the entity `alu_cell`, then the block configuration could use the declaration shown below.

```
for data_path
   for data_alu : alu
      use configuration project_cells.alu_struct
         generic map (width => 32)
                  port map (function_code => function, operand1 =>
                     op1, operand2 => op2,
                     result => result, flags => open);
   end for;
   other configuration items
end for;
```

B.6.3 Detailed Design Example

To illustrate the overall structure of a design description, a detailed design file for the example in Section 1.4 is shown in Figure5-6. The design file contains a number of design units which are analyzed in order. The first design unit is the entity declaration of count2. Following it are two secondary units, architectures of the count2 entity. These must follow the entity declaration, as they are dependent on it. Next is another entity declaration, this being a test bench for the counter. It is followed by a secondary unit dependent on it, a structural description of the test bench. Following this is a configuration declaration for the test bench. It refers to the previously defined library units in the working library, so no library clause is needed. Notice that the count2 entity is referred to in the configuration as work.count2, using the library name. Lastly, there is a configuration declaration for the test bench using the structural architecture of count2. It uses two library units from a separate reference library, misc. Hence a library clause is included before

the configuration declaration. The library units from this library are referred to in the configuration as misc.t_fliptlop and misc.inverter.

This design description includes all of the design units in one file. It is equally possible to separate them into a number of files, with the opposite extreme being one design unit per file. If multiple files are used, you need to take care that you compile the files in the correct order, and re-compile dependent files if changes are made to one design unit. Source code control systems can be of use in automating this process.

```
-- primary unit: entity declaration of count2

entity count3 is
    generic (prop_delay: Time: = 10 ns);
    port (clock: T0, T1, T2: in bit;
          Q0,Q1,Q2: out bit);
end count3;

-- secondary unit: a behavioral architecture body of count3

architecture behavior of count3 is

begin

    count_up: process (clock)

       variable count_value : natural :=0;

    begin

       if clock ='1' then

          count_value :=(count_value + 1) mod 4

          Q0<=bit'val(count_value mod 2) after prop_delay

          Q1<=bit'val(count_value mod 2) after prop_delay

          Q2<=bit'val(count_value mod 2) after prop_delay

       end if;

    end process count_up;

end behavior;

-- secondary unit: a structural architecture body of count3

architecture structure of count3 is
    component T_flip_flop
       port (CLK: in bit; Q: out bit;
     end component;

    component AND_gate
```

```
        port (A, B: in bit; C: out bit;
    end component;

    signal C, D, E, FF0, FF1, FF2; bit;

begin
    bit_0: T_flip_flop port map (CLK => Clock,
                                 Q0 => FF0, Q1=>FF1, Q2=>FF2);
    AND: AND_gate port map (A=>FF1, B=>FF0, C=> (A and B));

    bit_1: T_flip_flop port map (CLK => Clock,
                                 Q0 => FF0, Q1=>FF1, Q2=>FF2);

    Q0<=FF0;
    Q1<=FF1;
    Q2<=FF2;

end structure;

--primary unit: entity declaration of test bench

entity test_count3 is

end test_count3;

-- secondary unit: structural architecture body of test bench

architecture structure of test_count3 is

    signal clock, QO, Q1, Q2 : bit;

    component count3

       port (clock : in bit;

             Q0, Q1, Q2 : out bit);

    end component;

begin

    counter: count3

       port map (clock => clock, QO => QO, Q1 => Q1, Q2 => Q2):

    clock_driver: process

    begin

       clock <= '0', '1' after 50 ns;

       wait for 100 ns;

    end process clock_driver;

end structure;

-- primary unit: configuration using behavioral architecture
```

```
configuration test_count3_behavior of test_count2 is
    for structure  -- of test_count3
       for counter : count3
          use entity work.count3(behavior);
       end for;
    end for;
end test_count3_behavior;
-- primary unit: configuration using structural architecture
library misc;
configuration test_count3_structure of test_count3 is
    for structure  -- of test_count3
       for counter: count3
          use entity work.count3(structure);
          for structure  -- of count_3
             for all : T_flip_flop
                use entity misc.T_flip_flop(behavior);
             end for;
             for all : AND_gate
                use entity misc.AND_gate(behavior);
             end for;
          end for;
       end for;
    end for;
end test_count3_structure;
```

Appendix C

Introduction to Verilog

This appendix provides a brief overview of the Verilog Hardware Description Language (HDL). Like VHDL which we introduced in Appendix B, Verilog is a programming language used to describe a digital system and its components. Our discussion focuses on the use of Verilog HDL at the architectural or behavioral levels, and design at the Register Transfer Level (RTL).

C.1 Description

Verilog HDL is a Hardware Description Language (HDL). A Hardware Description Language is a language used to describe a digital system, such as a CPU or components such as an ALU. Moreover, a digital system may be described at several levels. For instance, an HDL might describe the layout of the interconnecting wires and components such as gates and flip flops. Verilog can also be used to describe the registers and the transfers of vectors of information between registers. This is called the *Register Transfer Level (RTL)*[*]. Verilog supports all of these levels. However, this handout focuses on only the portions of Verilog which support the RTL level.

Both Verilog and VHDL are used extensively in industry and academia. While there are many similarities between these two programming languages, Verilog resembles the C programming language whereas VHDL resembles the Ada programming language.

Verilog[†] was introduced in 1985 by Gateway Design System Corporation, now a subsidiary of Cadence Design Systems. Originally, Verilog HDL was a proprietary language of Cadence, but now it is open in the public domain. Verilog HDL allows a hardware designer to describe designs at a high level of abstraction such as at the architectural or behavioral level as well as the lower implementation levels (i.e., gate and switch levels) leading to Very Large Scale Integration (VLSI) Integrated Circuits (IC) layouts and chip fabrication. Like VHDL, Verilog is used in the simulation of designs prior to fabrication.

---

* *Register transfer level description (RTL), also called register transfer logic is a description of a digital electronic circuit in terms of data flow between registers, which store information between clock cycles in a digital circuit. The RTL description specifies what and where this information is stored and how it is passed through the circuit during its operation. RTL design is the design of an RTL description of a system. The term is also used synonymously with "RTL description". RTL is used in the logic design phase of the IC design cycle. Logic simulator tools may verify the correctness of the design by simulating its functionality using its RTL description, among other things. Logic synthesis tools may be used to automatically convert the RTL description of a digital system into a gate level description of the system.*

† *Another programming language, known as VeriWell, is an implementation of Verilog introduced by Wellspring Solutions, Inc. in 1992.*

C.2 Verilog Applications

As we've learned in Chapter 11, present-day digital systems are highly complex and are available in CPLD and FPGA technology. Accordingly, digital circuit design engineers must implement their designs with hardware description languages (HDLs) prior to fabrication. The most prominent modern HDLs in industry are Verilog and VHDL.

Verilog allows hardware designers to express their design with *behavioral constructs*, putting aside the details of implementation to a later stage of design in the design. Even though a behavioral construct is a high level description of a digital system, it is still a precise notation.

C.3 The Verilog Programming Language

Verilog describes a digital system as a set of modules. Comments begin with the symbol "//" and terminate at the end of the line or by /* to */ across several lines. Keywords, e.g., *module*, are reserved and in all lower case letters. The language is case sensitive, meaning upper and lower case letters are different. Spaces are important. A simple example is given below.

Example C.1

Using Verilog, design the behavioral model of a 2-input NAND gate.

Solution:

```
// Behavioral model of a 2-input NOR gate
// Designer's name and date
module NOR (in1, in2, out);
input in1, in2;
output out;
// Continuous assign statement follows
assign out = ~(in1 | in2);
endmodule
```

In Example C.1 above, the lines with `''//''` are comment lines, and **`module`** is the behavioral specification of module NOR with ports (inputs and output) **`in1`** and **`in1`** and **`out`**. The continuous assignment **`assign`** monitors continuously for changes to variables in its right hand side and whenever that happens the right hand side is re-evaluated and the result immediately propagated to the left hand side (**`out`**). The continuous assignment statement is used to model combinational circuits where the outputs change whenever there is a change in the inputs. The symbol `''|''` is the bit-wise OR operator in Verilog.

Example C.2

Using Verilog, design the behavioral model of a 2-input OR gate using two 2-input NOR gates.

Solution:

A 2-input OR gate can be formed by connecting the output of one NOR gate to both inputs of the other. This module has two instances of the NOR module of Example C.1, the first referred to as NOR1, and the second as NOR2 connected together by an internal wire w1.

```
// Behavioral model of a 2-input OR gate using NOR gates
// Designer's name and date
module OR (in1, in2, out);
input in1, in2;
output out;
wire w1;
// two instantiations of the module NOR, Example C.1
NOR NOR1(in1, in2, w1);
NOR NOR2(w1, w1, out);
// Continuous assign statement follows
assign out = (in1 | in2);
endmodule
```

The general form to invoke an instance of a module is:

```
<module name> <parameter list> <instance name> (<port list>);
```

where <parameter list> are values of parameters passed to the instance. An example parameter passed would be the delay for a gate.

The following example is a simple Register Transfer Level (RTL) digital model.

Example C.3

A digital circuit consists of two 8-bit registers A and B and one 1-bit register (flip flop) C. Initially, the contents of Register A are all zeros, while the contents of Registers B and C are not known. Register A behaves as a BCD counter starting the count from 0 and stops at 9 after 27 simulation units. Register A is incremented by one at the first, fourth, seven, and so on simulation units, the contents of Register B are changed during the second, fifth, eighth, and so on simulation units, and the content of Register C is updated during the third, sixth, ninth, and so on simulation units. The last four bits of Register B are set to "NOT" (ones compliment) of the last four bits of Register A, and C forms the ANDing of the last two bits of Register A. Implement this digital circuit using

Verilog.

Solution:

We start with Table C.1. and since we do not know the initial contents or Registers B and C, we denote them with don't cares. Changes are shown in bold type.

TABLE C.1 Table for Example C.3

Time (Simulation Unit)	Register A	Register B	Register C
0	0000 0000	xxxx xxxx	x
1	0000 **0001**	xxxx xxxx	x
2	0000 0001	xxxx **1110**	x
3	0000 0001	xxxx 1110	**0**
4	0000 **0010**	xxxx 1110	0
5	0000 0010	xxxx **1101**	0
6	0000 0010	xxxx 1101	**0**
7	0000 **0011**	xxxx 1101	0
8	0000 0011	xxxx **1100**	0
9	0000 0011	xxxx 1100	**1**
10	0000 **0100**	xxxx 1100	1
11	0000 0100	xxxx **1011**	1
12	0000 0100	xxxx 1011	**0**
13	0000 **0101**	xxxx 1011	0
14	0000 0101	xxxx **1010**	0
15	0000 0101	xxxx 1010	**0**
16	0000 **0110**	xxxx 1010	0
17	0000 0110	xxxx **1001**	0
18	0000 0110	xxxx 1001	**0**
19	0000 **0111**	xxxx 1001	0
20	0000 0111	xxxx **1000**	0
21	0000 0111	xxxx 1000	**1**
22	0000 **1000**	xxxx 1000	1
23	0000 1000	xxxx **0111**	1
24	0000 1000	xxxx 0111	**0**
25	0000 **1001**	xxxx 0111	0
26	0000 1001	xxxx **0110**	0
27	0000 1001	xxxx 0110	**0**
Stop			

We call our digital circuit `rtl`.

```
//RTL model in Verilog
```

```
module rtl;
// Register A is incremented by one.  Then last four bits of B is
// set to "not" of the last four bits of A.  C is the "ANDing"
// of the last two bits in A.
// declare registers A, B, and C
reg [0:7] A, B;
reg       C;
// The two "initial"s and "always" below will run concurrently
initial begin: stop_at
   // Will stop the execution after 28 simulation units.
#28; $stop;
end
// The statements below begin at simulation time 0 since no #k is
// specified
initial begin: Init
// Initialize register A.  Other registers have values of "x"
A = 0;
// Display a header
$display("Time   A         B    C");
// Prints the values anytime a value of A, B or C changes
$monitor("  %0d %b %b %b", $time, A, B, C);
end
//main_process will loop until simulation is over
always begin: main_process
// #1 means do after one unit of simulation time
#1 A = A + 1;
#1 B[4:7] = ~A[4:7]; // ~ is bitwise "not" operator
#1 C = &A[6:7];      // bitwise "ANDing" of last 2 bits of A
end
endmodule
```

In module **simple**, we declared A and B as 8-bit registers and C a 1-bit register or flip-flop. Inside

of the module, the one "always" and two "initial" constructs describe three threads of control, i.e., they run at the same time or **concurrently**. Within the **initial** construct, statements are executed sequentially much like in C or other traditional imperative programming languages. The **always** construct is the same as the **initial** construct except that it loops forever as long as the simulation runs.

The notation #1 means to execute the statement after delay of one unit of simulated time. Therefore, the thread of control caused by the first initial construct will delay for 28 time units before calling the system task $stop and stop the simulation. The $display system task allows the designer to print a message. Every time unit that one of the listed variables' value changes, the $monitor system task prints a message. The system function $time returns the current value of simulated time.

Upon completion of the program, Verilog displays Table C.1.

C.4 Lexical Conventions

The *lexical conventions* in Verilog are very similar to those of C++. Verilog consists of a stream of *tokens** separated by *delimiters*.† As stated before, comments are designated by // to the end of a line or by /* to */ across several lines. Keywords, e.g., **module**, are reserved and in all lower case letters. The language is case sensitive, meaning upper and lower case letters are different. Spaces are important.

Numbers in Verilog are specified in the traditional form of a series of digits with or without a sign but also in the following form:

```
<size><base format><number>
```

where <size> contains decimal digits that specify the size of the constant in the number of bits. The <size> is optional. The <base format> is the single quotation (‘) followed by one of the following characters b, d, o and h, for binary, decimal, octal, and hexadecimal respectively. The <number> part contains digits which are legal for the <base format>.

Examples:

```
‘b1011      // binary number 00001011
‘o4567      // octal number
9867        // decimal number
```

* *Tokens are comments, operators, numbers, strings, identifiers, and keywords.*

† *Delimiters are white spaces (blank spaces, tabs, or new lines), commas, left and right parentheses, and semicolons. The semicolon is used to indicate the end of a statement.*

```
'hA9F7      // hexadecimal number
8'b1011     // 8-bit binary number 00001011
-8'b101     // 8-bit binary word in twos complement, that is,
               00000101
6'd54       // 6-character decimal number, that is, 540000
4'b1x10     // 4-bit binary number with third position from right a
               don't care.
```

The sign plus (+) or minus (–) must always be in front of the number. Of course the plus sign may be omitted. The <**number**> part must not contain a sign.

A *string* in Verilog is a sequence of characters enclosed in double quotes. For example,

```
"this is a string"
```

Operators are one, two or three characters and are used in expressions. We will discuss operators in Section C.7.

Identifiers in Verilog are specified by a letter or underscore followed by zero or more letters, digits, dollar signs and underscores. Identifiers can be up to 1024 characters.

C.5 Program Structure

As stated earlier, Verilog describes a digital system as a set of modules. Each of these modules has an interface to other modules to describe how they are interconnected. Usually, we place one module per file but this is not a requirement. The modules may run concurrently, but usually we have one top level module which specifies a closed system containing both test data and hardware models. The top level module invokes instances of other modules.

Modules can be used to describe hardware ranging from simple gates to complete systems, e.g., a microprocessor. Modules can either be specified behaviorally or structurally (or a combination of the two). A *behavioral specification* in Verilog defines the behavior of a digital system (module) using traditional programming language constructs such as, **if** assignment statements. A *structural specification* in Verilog expresses the behavior of a digital system (module) as a hierarchical interconnection of smaller modules. At the bottom of the hierarchy the components must be primitives or those specified behaviorally. Verilog *primitives* include gates, e.g., nand, as well as pass transistors (switches).

The structure of a module is the following:

```
module <module name> (<port list>);
<declares>
<module items>
endmodule
```

The <module name> is an identifier that uniquely assigns the module name. Once assigned, a module is never invoked within a program. The <port list> is a list of input, inout, and output ports which are used to connect to other modules. The <declares> part specifies data objects as registers, memories, and wiring interconnections, and procedural constructs such as functions and tasks.
The <module items> may be initial constructs, always constructs, continuous assignments or instances of modules.

As we've learned in Examples C.1 and C.2, continuous assignments use the keyword **assign**. However, *procedural assignments* have the form

```
<reg variable> = <expression>
```

where the <reg variable> must be a register or memory. Procedural assignment must only appear in **initial** and **always** constructs as in Example C.3. The **initial** and **always** constructs are used to model sequential logic circuits.

Example C.4

Define a module that sets test data and sets the monitoring of variables for a 2-input OR gate using two 2-input NOR gates as in Example C.2.

Solution:

For this module we need to use two 1-bit registers to hold the values of the two inputs to the first NOR gate, denoted as a and b. Our module is described as follows:

```
module test_OR;// Module to test and set up monitoring of variables
reg a, b;
wire out1, out2;
initial begin  // Test data
   a = 0;  b = 0;
   #1 a = 1;
   #1 b = 1;
   #1 a = 0;
end
initial begin // Set up monitoring
   $monitor("Time=%0d a=%b b=%b out1=%b out2=%b",
         $time, a, b, out1, out2);
end
```

```
// Instances of modules OR and NOR
   NOR  gate1(a, b, out1);
   OR gate2(a, b, out2);
endmodule
```

The statements in the block of the first **initial** construct will be executed sequentially, some of which are delayed by #1, i.e., one unit of simulated time.

The simulator produces the following output.

```
Time=0 a=0 b=0 out1=1 out2=0
Time=1 a=1 b=0 out1=1 out2=0
Time=2 a=1 b=1 out1=0 out2=1
Time=3 a=0 b=1 out1=1 out2=0
```

We recall that the **always** construct behaves the same as the **initial** construct except that it loops forever. Since the **always** construct was not used in this example, the simulator ran out of events, and we did not need to explicitly stop the simulation as we did in Example C.3 where we used the statement `#28; $stop`.

C.6 Data Types

The *data types* in Verilog are classified as physical data types, and abstract data types. These are described in the following subsections.

C.6.1 Physical Data Types

The *physical data types* used in Verilog are those used to model digital circuits. The primary types are for modeling flip flops and registers (**reg**) and wires (**wire**). The latter can include the types wired- AND (**wand**), wired-OR (**wor**) and tri-state devices (**tri**). The **reg** variables store the last value that was assigned to them whereas the **wire** variables represent physical connections between structural entities such as gates. A **wire** does not store a value; this variable is essentially a label for a wire.

The reg and wire data objects may have the following possible values:

0 logical zero or false

1 logical one or true

x unknown (don't care) logical value

z high impedance of tristate device

The **reg** variables are initialized to **x** at the start of the simulation. Any wire variable not connected to something has the **x** value.

We specify the size of a register or wire in the declaration.

Examples:

```
reg [0:15] A, B; // Registers A and B are 16 bits long where the
                    most significant bit is the zeroth position bit

reg [7:0] C;     // Register C is 8 bits long where the most
                    significant bit is the seventh position bit

wire [0:3] Dataout;// Dataout is 4 bits long where the most
                      significant bit is the zeroth position bit
```

The bits in a register or wire can be referenced by the notation

```
[<start-bit>:<end-bit>]
```

Example:

Consider the statements where the symbol (|) implies the OR operation and the contents inside the brackets ({}), separated by commas, imply concatenation. Register B is also 8 bits long.

```
initial begin: int1

A = 8'b10010011;

B = {A[0:3] | A[4:7], 4'b0101};

end
```

Here, Register B is set to the last four bits (0011) of Register A, and it is bitwise ORed with the first four bits (1001) of Register A. It is then concatenated with 0101. Thus, after these statements are executed, the content of Register B is 10110101.

An *argument* may be replicated by specifying a repetition number of the form:

```
{repetition_number{exp1, exp2, ... , expn}}
```

Example:

```
C = {2{4'b1001}}; //C assigned the bit vector 8'b10011001
```

The range referencing in an expression must have constant expression indices. However, a single bit may be referenced by a variable.

Examples:

```
reg [0:7] A, B;

B = 3;

A[0: B] = 3'b111; // ILLEGAL (B) – indices MUST be constants

A[B] = 1'b1;      // A single bit reference is LEGAL
```

Memories are specified as vectors of registers.

Example:

```
reg [31:0]  Mem [0:1023];
```

Here, **Mem** consists of 1K (1024) words each 32-bits long. The notation `Mem[0]` references the zeroth word of memory.

C.6.2 Abstract Data Types

Abstract data types are data types which do not have a corresponding hardware realization. An integer variable used to count the number of times an event occurs, is a good example of an abstract data type. These data types include **integer**, **real**, and **time**. We should remember that a **reg** variable is unsigned and that an integer variable is a signed 32-bit integer. The data type **time** variables hold 64-bit quantities and are used in conjunction with the **$time** system function. Arrays of **integer** and **time** variables (but not reals) are allowed. However, **reals** are not allowed. Multiple dimensional arrays are not allowed either.

Examples:

```
integer Count;      // simple signed 32-bit integer
integer K[1:64];    // an array of 64 integers
time Start, Stop;   // Two 64-bit time variables
```

C.7 Operators

Operators in Verilog are classified as binary arithmetic operators, unary arithmetic operators, relational operators, logical operators, bitwise operators, unary reduction operators, and conditional operators. We will discuss these in the following subsections and we will list the operator precedence at the last subsection.

C.7.1 Binary Arithmetic Operators

Binary arithmetic operators operate on two operands. Register and net (wire) operands are treated as unsigned. However, real and integer operands may be signed. If any bit of an operand is unknown ('x') then the result is unknown. They are listed in Table C.2.

TABLE C.2

Symbol	Operation	Comments
+	Addition	
–	Subtraction	
*	Multiplication	
/	Division	Division by 0 results in an x
%	Modulus	

C.7.2 Unary Arithmetic Operators

A unary operator is an operator that takes only one operand- for example, unary minus (as in –1.5).

TABLE C.3 Unary operator

Symbol	Operation	Comments
–	Unary minus	Changes the sign of its operand

C.7.3 Relational Operators

Relational operators compare two operands and return a logical value, i.e., TRUE (1) or FALSE (0). If any bit is unknown, the relation is ambiguous and the result is unknown. They are shown in Table C.4.

TABLE C.4 Relational operators in Verilog

Symbol	Operation	Comments
==	Equality	Logical equality
!=	Inequality	Logical inequality
>	Greater than	
>=	Greater than or equal	
<	Less than	
<=	Less than or equal	

C.7.4 Logical Operators

*Logical operators** operate on logical operands and return a logical value, i. e., TRUE(1) or FALSE(0). They are used in **`if`** and **`while`** statements. They are listed in Table C.5.

TABLE C.5 Logical operators in Verilog

Symbol	Operation	Comments
!	Negation	Logical negation
&&	AND	Logical AND
||	OR	Logical OR

C.7.5 Bitwise Operators

Bitwise operators operate on the bits of the operand or operands. For example, the result of A & B is the AND of each corresponding bit of A with B. Operating on an unknown (x) bit results in

* *We must not confuse logical operators with the bitwise Boolean operators. The logical operator* ! *is a logical* NOT *while the operator* ~ *is a bitwise* NOT. *The logical operator is a negation, for instance,* !(7 == 4) *is* TRUE. *The bitwise* NOT *complements all bits, for instance,* ~{1,0,0,1} *is equivalent to* 0110.

the expected value. For example, the AND of an x with a FALSE is an FALSE. The OR of an x with a TRUE is a TRUE. They are listed in Table C.6.

TABLE C.6 Bitwise operators in Verilog

Symbol	Operation	Comments
~	Negation	Bitwise equality
&	AND	Bitwise AND
|	OR	Bitwise OR
^	XOR	Bitwise XOR
~&	NAND	Bitwise NAND
~|	NOR	Bitwise NOR
~^	XNOR	Bitwise XNOR (Equivalence)

C.7.6 Unary Reduction Operators

Unary reduction operators produce a single bit result from applying the operator to all of the bits of the operand. Reduction is performed left (first bit) to right. Thus, `&A` will `AND` all the bits of `A`.

Examples:

```
Unary reduction AND (&):
   &4'b1111 results in 1'b1 // (1&1&1&1) = 1
   &2'bx1 results in 1'bx   // x&1 = x
   &2'bz1 results in'bx     // z is treated as x for all unary operators
Unary reduction NAND (~&):
   ~&4'b1111 results in 1'b0
   ~&2'bx1 results in 1'bx
```

The unary reduction operators are shown in Table C.7.

TABLE C.7 Unary reduction operators in Verilog

Symbol	Operation	Comments
&	AND	AND unary reduction
|	OR	OR unary reduction
^	XOR	XOR unary reduction
~&	NAND	NAND unary reduction
~|	NOR	NOR unary reduction
~^	XNOR	XNOR unary reduction

C.7.7 Other Operators

Other operators in Verilog are those not listed in Tables C.2 through C.7. They are listed in Table C.8.

TABLE C.8 Other operators in Verilog

Symbol	Operation	Comments
`===`	Equality	Bitwise comparison between x and y binary words. For equality, all bits must match. Returns True or False.
`!==`	Inequality	Bitwise comparison between x and y binary words. Any difference results in inequality. Returns True or False.
`{a,b,..,n}`	Concatenation	Concatenates bits of comma separated words, e.g., a, b, ..,n
<<	Shift left	After left shift, vacated positions are filled with zeros, e.g., X=X<<4 shifts four bits to the left with zero fills.
>>	Shift right	After right shift, vacated positions are filled with zeros, e.g., Y=Y>>3 shifts three bits to the right with zero fills.
`?:`	Conditional[a]	Assigns one of two possible values depending on the conditional expression

a. *Operates as in language C. For example, X=Y>Z ? A+2 : A–3 implies that if Y>Z, the value of X is A+2, otherwise, the value of X is A–3.*

C.7.8 Operator Precedence

The *precedence of operators* is shown in Table C.9, listed from highest (top row) to lowest (bottom row). Operators on the same line have the same precedence and are executed from left to right in an expression. Parentheses can be used to change the precedence or clarify the situation.

TABLE C.9 Operator Precedence in Verilog

Symbol(s)	Operation	Precedence
! & ~& | ~| ^ ~^ + –	Unary	Highest
* / %	Binary arithmetic	↓
+ –	Binary arithmetic	↓
<< >>	Shift left, Shift right	↓
< <= > >=	Relational	↓
== != === ~==	Relational, Other	↓
& ~& ^ ~^	Bitwise	↓
| ~|	Bitwise	↓
&&	Logical	↓
||	Logical	↓
?:	Conditional	Lowest

The use of parentheses to improve readability is highly recommended.

C.8 Control Statements

Control statements in Verilog are classified as selection, and repetition. They are discussed in the subsections below. Their description is best illustrated with examples.

C.8.1 Selection Statements

Selection statements use the **if**, **else**, and **case**, **endcase** statements.

Example:

```
if (X == 8)
   begin
      A = 4;
   end
else
   begin
      A = 16;
   end
```

The **case**, and **endcase** statements have the following syntax:

```
case (<expression>)
   <value1>: <statement>
   <value2>: <statement>
   default: <statement>
endcase
```

The first <value> that matches the value of the <expression> is selected and the appropriate statement is executed. Then, control is transferred to after the endcase.

Example:

```
case (sign)
   1'bz: $display("Zero, number is positive");
   1'bx: $display("One, number is negative");
   default: $display("Signal is %b", sign);
endcase
```

This example checks the most significant bit of a binary word for its sign, 0 for +, and 1 for –.

C.8.2 Repetition Statements

Repetition statements use the **for**, **while**, and **repeat** loops.

The **for** loops allow a group of statements to be repeated a fixed, predetermined number of times. As in other programming languages, **for** specifies an initial value, e.g., `i = 0`, a range of values, e.g., `i < k`, and the increment statement, e.g., `i = i + 1`.

Example:

```
for(n = 0; n < 15; n = n + 1)
   begin
      $display("n= %0d", n);
   end
```

The **while** loops repeat statements as long as a logical expression is true.

Example:

```
n = 0;
while(n < 20)
   begin
      $display("n= %0d", n);
      n = n + 1;
   end
```

The **repeat** loops repeat statements for a fixed number of times. In the example below, it repeats 10 times.

```
repeat (10)
   begin
      $display("n= %0d", n);
       n = n + 1;
   end
```

C.9 Other Statements

Other statements in Verilog are classified as parameter, continuous assignment, blocking procedural assignments, and non-blocking procedural assignments. They are discussed in the subsections below. Their description is best illustrated with examples.

C.9.1 Parameter Statements

The parameter statement allows us to give a constant a name. Typical uses are to specify width of registers and delays. For example,

```
parameter word_size = 16;
```

is a parameter statement that specifies a binary word size.

C.9.2 Continuous Assignment Statements

As we indicated in Examples C.1 and C.2, *continuous assignments* are used with **`wire`** variables and are evaluated and updated whenever an input operand changes value. For instance, the continuous assignment

```
assign out = ~(in1 & in2);
```

performs the ANDing of the wires `in1` and `in2`. We recall that **`assign`** is used to differentiate between continuous assignment from procedural assignments.

C.9.3 Blocking Assignment Statements

Blocking assignment statements are specified with the = operator and are executed in the order they are specified in a sequential block. A blocking assignment will not block execution of statements in a parallel block (another block). The following statements specify blocking assignments only.

```
reg X, Y, Z;
reg [15:0] reg_A, reg_B;
                          // Behavioral statements must be inside
                          // an initial or an always block.
initial
begin
    X = 0;                          // Scalar assignments
    Y = 1;
    Z = 1;
    reg_A = 16'b0;                  // Initialize vectors
    reg_B = reg_A;                  // Vector assignments
    #15 reg_A[2] = 1 'b';           // Bit assignment with delay
    #10 reg_B[15:13] = {X, Y, Z}; // Assign result of concatenation
                                    // to part of a vector
end
```

If blocking statements are used, the order of execution in a particular block is sequential. Thus, in the block above, block, all statements `X = 0` through `reg_B = reg_A`, are executed sequentially at time zero. The statement `reg_A[2] = 1` is executed at time = 1, and the statement `reg_B[15:13] = {X, Y, Z}` is executed at time = 25, that is, 10 time units after `reg_A[2] = 1`. In other words, `reg_B[15:13] = {X, Y, Z}` is blocked until `reg_A[2] = 1` is executed.

In an assignment, the result of right-hand expression is left truncated or left zero filled to match the length of the left-hand side.

C.9.4 Non-Blocking Assignment Statements

The *non-blocking assignment statements* allow scheduling of assignments without blocking execution of the statements that follow in a sequential block. Non-blocking assignment statements are specified with the <= operator. Even though this operator has the same symbol as a relational operator, less_than_equal_to, the different operations will be clear from the context.

The following statements specify both blocking and non-blocking assignments.

```
reg X, Y, Z;
reg [15:0] reg_A, reg_B;
                          // Behavioral statements must be inside
                          // an initial or an always block.
initial
begin
    X = 0;                          // Scalar assignments
    Y = 1;
    Z = 1;
    reg_A = 16'b0;                  // Initialize vectors
    reg_B = reg_A;                  // Vector assignments
    #15 reg_A[2] <= 1 'b';          // Bit assignment with delay
    #10 reg_B[15:13] <= {X, Y, Z};// Assign result of concatenation
                                    // to part of a vector
end
```

In the example above, the statements `X = 0` through `reg_B = reg_A` are executed sequentially at time 0 (blocking). The two non-blocking assignments are processed at the same simulation time, but `reg_A[2] <= 1` is scheduled to execute after 15 time units (i.e., time = 15) and `reg_B[15:13] <= {X, Y, Z}` is scheduled to execute after 10 time units (i.e., time = 10).

The simulator schedules a non-blocking assignment statement to execute and continues to the next statement in the block without waiting for the non-blocking statement to complete execution.

C.10 System Tasks

System tasks in Verilog are standard tasks to perform certain subprograms and routines. These tasks are identified with the format

```
$<keyword>
```

Operations such as displaying on the screen, monitoring values of nets, opening a file, writing a file, and reading a file into an array are done by system tasks.

Examples:

```
$display(in1,in2,out1,out2);  // in1, in2, out1, out2 can be quoted
                              // as strings, variables, or expressions

$display(''This counter..'');// Display the string in quotes

$display($time); // Display value of current simulation time 120
```

The last statement above displays the value 120. This statement also illustrates the systems task **$time**.

The examples below illustrate the **$monitor** system task.

```
$display(in1,in2,out1,out2);  // in1, in2, out1, out2 can be quoted
                              // as strings, variables, or expressions

$monitoron;

$monitoroff;
```

Two last two tasks are used to switch monitoring on and off.

To enter information into a systems file we must first declare an integer variable to identify the file; then we must open the file, and, as a last step, we must write information to the file. To open a file, the format is

```
<internal_file_name> = $fopen (''<system_file _name>'')

integer file001    //Declare an internal integer name for the file

file001 = $fopen(''Four_bit_adder.out'');// Open an output file
```

To write information to a file, we use the format

```
$fdisplay(<internal_ file_name>, <format list>, <variable_list>)
```

The `internal_file_name` is an integer variable that must be declared in the declarations. The format list is a list of formats for the variables separated by commas. The `variable_list` is a list of variables to be outputted separated by commas.

Example:

```
$fdisplay(file001, ''%d %d %d %d %d'', in1, in2, out1, out2);
```

The commonly used format specifiers are

```
%b display in binary format
%c display in ASCII character format
%d display in decimal format
%h display in hex format
%o display in octal format
%s display in string format
```

To load an array (memory) from a systems file, we use the format

```
$fmemreadh(î\<file_name>, <array_name>);
$fmemreadb(î\<file_name>, <array_name>);
```

The first format is for reading from a hexadecimal file; the second for reading from a binary file.

Example:

```
$readmemh(''count_test_input.hex'', mem);
```

The file to be read must have form

```
@address_in_decimal
data0 data1 data2 ...
...
```

The address is the starting address to load. The data are the values to be loaded either in hexadecimal or binary as specified in the read.

Other systems tasks are listed below. For a complete list, please refer to the Verilog Language Reference Manual.

$stop – sets the simulator into a halt mode, issues an interactive command prompt, and passes control to the user.

$settrace – enables tracing of simulation activity. The trace consists of various information, including the current simulation time, the line number, the file name, module and any results from executing the statement. we can turn off the trace using the **$cleartrace** system task listed below.

$cleartrace – turns off the trace. See **$settrace** system task above to set the trace.

$scope – allows the user assign a particular level of hierarchy as the interactive scope for identifying objects. It is useful during debugging as the user may change the scope to inspect the values of variables in different modules, tasks, and functions.

$showscopes – displays a complete lists of all the modules, tasks, functions and named blocks that are defined at the current scope level.

$showvars – produces status information for register and net (wires) variables, both scalar and vector. It displays the status of all variables in the current scope. When invoked with a list of variables, it shows only the status of the specified variables.

C.11 Functions

Functions in Verilog behave as subprograms in other languages. The purpose of a function is to return a value that is to be used in an expression. A function definition must contain at least one **input** argument, and must not contain parameter lists or parentheses. The definition of a function is the following:

```
function <range or type> <function name>;
   <argument ports>
   <declarations>
   <statements>
endfunction
```

where `<range or type>` is the type of the results referenced back to the expression where the function was called. Inside the function, we must assign the function name a value.

Example:

```
module functions;
   function [1:1] one_bit_adder;   // function definition
    input a0, b0;                  // two input argument ports
    reg A;                         // register declaration
    begin
       A = 1;
       if (a0 == b0)
          one_bit_adder = 1 & R;
       else
          one_bit_adder = 0;
    end
endfunction
```

C.12 Timing Control

Timing control in Verilog is used to specify the timing at which a procedural statement is to occur. We will discuss the different types in the following subsections.

C.12.1 Delay Control

A delay control statement specifies the time duration at which it executes. A delay control is identified with the symbol # which may be followed by an expression consisting of variables. For instance, the statement

```
#30 X=X+1
```

specifies a delay of 30 simulation time units before the contents of Register X are incremented.

C.12.2 Event Control

An *event* in Verilog is used to scheduled for discrete times. They are scheduled on a first-come, first-served basis. The simulator removes all the events for the current simulation time and processes them. During the processing, more events may be created and placed in the proper place in the queue for subsequent processing. When all the events of the current time have been processed, the simulator advances time and processes the next events in line.

Verilog uses the symbol @ to specify an event timing control. Statements can be executed on changes in a signal value or at a positive or negative transition of the signal value. The keyword **posedge** for a zero to one transition, and the keyword **negedge** for a one to zero transition.

Examples:

```
@(posedge clock) q = d;
@(negedge clock) q = d;
```

The first statement above is executed whenever the signal clock goes through a positive transition, and the second whenever it goes through a negative transition.

C.12.3 Wait Control

The *wait* control statement in Verilog allows a procedural statement or a block to be delayed until a condition becomes true.

Example:

```
wait (A == 3)
begin
    A = B&C;
end
```

Whereas the posedge and negedge statement are edge sensitive, the wait statement is level sensitive.

C.12.4 Fork and Join Control

The *fork* and *join* control statements in Verilog, allow us more than one thread of control inside an initial or always construct. For example, to have three threads of control, we can **fork** the thread into three and merge the three into one with a **join** as shown below

```
fork: three          //split thread into three; one for each begin-end
   begin
      // code for thread 1
   end
   begin
      // code for thread 2
   end
   begin
      // code for thread 3
   end
join                 // merge the three threads to one
```

In each statement between the **fork** and **join**, in this case, the three `begin-end` blocks, is executed concurrently. After all the threads are complete, the next statement after the **join** is executed. We must make sure that there is no conflict among the different threads. For instance, we should not change the state of a flip flop or register in two different threads during the same clock period.

Appendix D

Introduction to Boundary Scan Architecture

This appendix provides a brief overview of the boundary-scan architecture and the new technology trends that make using boundary-scan essential for the reduction in development and production costs. It also describes the various uses of boundary-scan and its application.

D.1 The IEEE Standard 1149.1

Boundary-scan, as defined by the IEEE Std. 1149.1 standard, is an integrated method for testing interconnects on printed circuit boards that is implemented at the IC level. The inability to test highly complex and dense printed circuit boards using traditional in-circuit testers and bed of nail fixtures was already evident in the mid eighties. Due to physical space constraints and loss of physical access to fine pitch components and BGA devices, fixtures cost increased dramatically while fixture reliability decreased at the same time.

D.2 Introduction

In the 1980s, the *Joint Test Action Group* (JTAG) developed a specification for boundary-scan testing that was standardized in 1990 as the IEEE Std. 1149.1-1990. In 1993 a new revision to the IEEE Std. 1149.1 standard was introduced (titled 1149.1a) and it contained many clarifications, corrections, and enhancements. In 1994, a supplement that contains a description of the *Boundary-Scan Description Language* (BSDL) was added to the standard. Since that time, this standard has been adopted by major electronics companies all over the world. Applications are found in high volume, high-end consumer products, telecommunication products, defense systems, computers, peripherals, and avionics. Now, due to its economic advantages, smaller companies that cannot afford expensive in-circuit testers are using boundary-scan.

The boundary-scan test architecture provides a means to test interconnects between integrated circuits on a board without using physical test probes. It adds a boundary-scan cell that includes a multiplexer and latches, to each pin on the device. Boundary-scan cells in a device can capture data from pin or core logic signals, or force data onto pins. Captured data are serially shifted out and externally compared to the expected results. Forced test data are serially shifted into the boundary-scan cells. All of this is controlled from a serial data path called the scan path or scan chain. The main elements of a boundary-scan device are shown in Figure D.1.

By allowing direct access to nets, boundary-scan eliminates the need for large number of test vectors, which are normally needed to properly initialize sequential logic. Tens or hundreds of vectors may do the job that had previously required thousands of vectors. Potential benefits realized from the use of boundary-scan are shorter test times, higher test coverage, increased diagnostic capability and lower capital equipment cost.

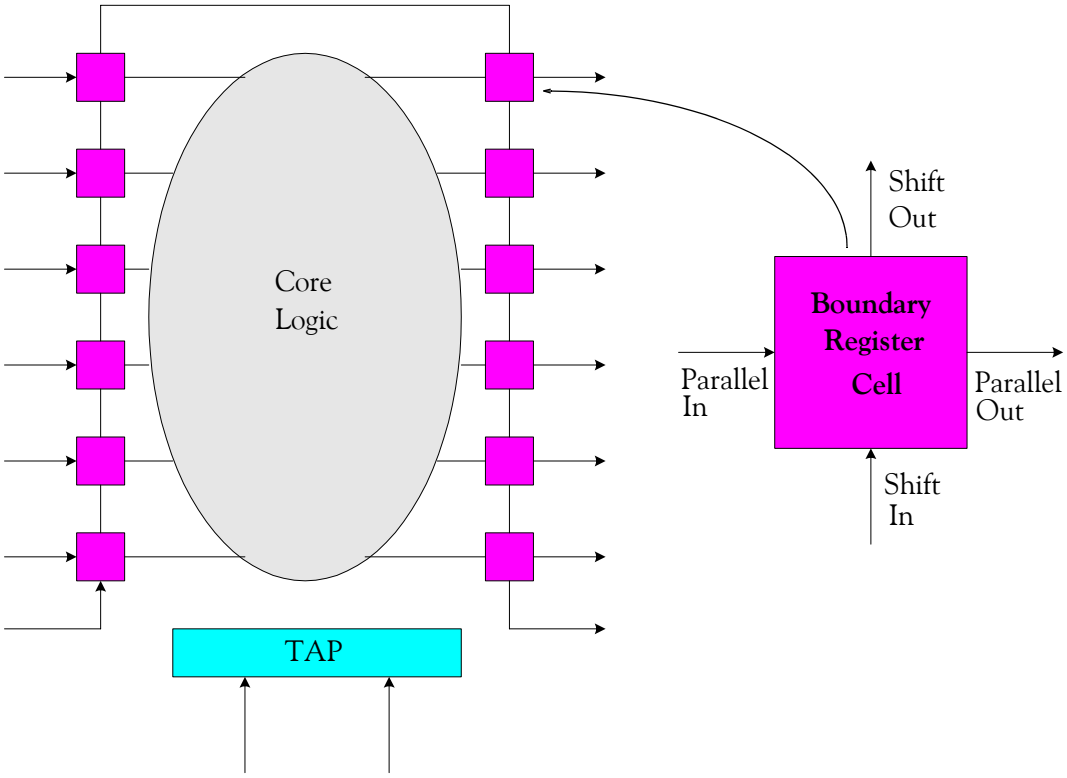


Figure D.1. The components of a boundary-scan device (Courtesy Corelis, Inc.)

The principles of interconnect test using boundary-scan are illustrated in Figure D.2, where two boundary-scan compliant devices, U1 and U2 interface with each other.

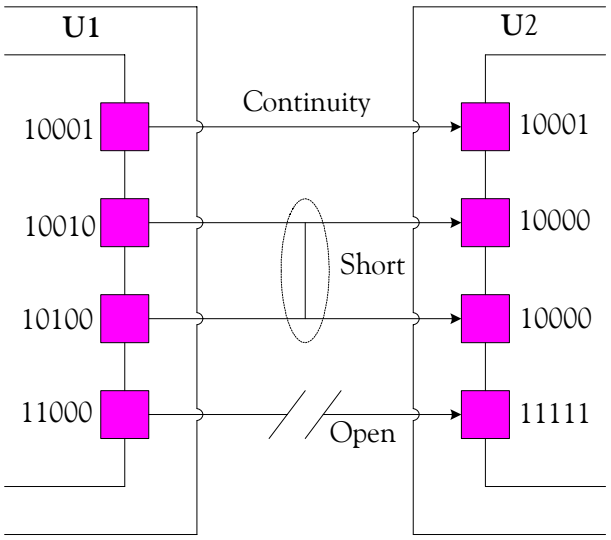


Figure D.2. Interconnect Test Example (Courtesy Corelis, Inc.)

In Figure D.2, Device U1 includes four outputs that are driving the four inputs of U2 with various values. We assume that the circuit includes a short between inputs and outputs 2 and 3, and an open between inputs and outputs 4. We also assume that a short between two nets behaves as a wired-AND and an open is sensed as logic 1. To detect and isolate the above defects, the tester is shifting into the U1 boundary-scan register the patterns shown in Figure D.2 and applying these patterns to the inputs of U2. The inputs values of U2 boundary-scan register are shifted out and compared to the expected results. In this case the results on inputs / outputs 2, 3, and 4, are not the same as the expected values and therefore the tester detects the faults on inputs / outputs 2, 3, and 4.

Boundary-scan tool vendors provide various types of stimulus and sophisticated algorithms that not only detect the failing circuits but also isolate the faults to specific devices, and pin numbers.

D.3 Boundary Scan Applications

The boundary-scan based testing was developed to be used in the production phase of a product, new developments and applications of the IEEE-1149.1 standard have enabled the use of boundary-scan in many other product life cycle phases. Specifically, boundary-scan technology is now applied to product design, prototype debugging and field services.

Recent technology advances for reduced consumer product size, such as portable phones and digital cameras, higher functional integration, faster clock rates, and shorter product life-cycle with dramatically faster time to market, has created new technology trends. However, the new technology has resulted in increased device complexity, and has created several problems in product development some of which are listed below.

- Most PCBs include components that are assembled on both sides of the board. Quite often, through-holes and traces are buried and are inaccessible.
- When the prototype arrives, a test fixture for the In-Circuit-Tester (ICT) is not available and therefore manufacturing defects can not be easily detected and isolated.
- Small-size products are not provided with test points and this makes it difficult or impossible to probe suspected nodes.
- Many Complex Programmable Logic Devices (CPLDs and FPGAs) and flash memory devices are not socketed; in many cases are soldered directly to the board.
- Whenever a new processor, CPLD, FPGA, or a different flash device is selected, one must learn how to program these devices.

Boundary-scan technology was developed to alleviate such problems. Some claim that the number of devices that include boundary-scan has grown exponentially. Almost every new microprocessor that is being introduced includes boundary-scan circuitry for testing and in-circuit emulation. Most of the CPLDs and FPGAs manufacturers such as Altera, Lattice, and Xilinx, to mention a few, have incorporated boundary-scan logic into their components including additional circuitry that uses the boundary-scan 4-wire interface to program their devices in-system.

Boundary-scan eliminates the problems listed above by providing several benefits some of which are listed below.

- Easy to implement Design For Testability (DFT) Rules.
- Testability report prior to PCB layout enhances DFT.
- Detection of possible packaging problems prior to PCB layout.
- No or little requirement for test fixtures and test points.
- Quickly diagnostics for detecting possible interconnect problems without any functional test code.
- Program code in flash devices, embedded design configuration data into CPLDs and FPGAs, and JTAG emulation and source-level debugging.

D.4 Board with Boundary-Scan Components

Figure D.3 shows a block diagram of a typical *boundary-scan board*.

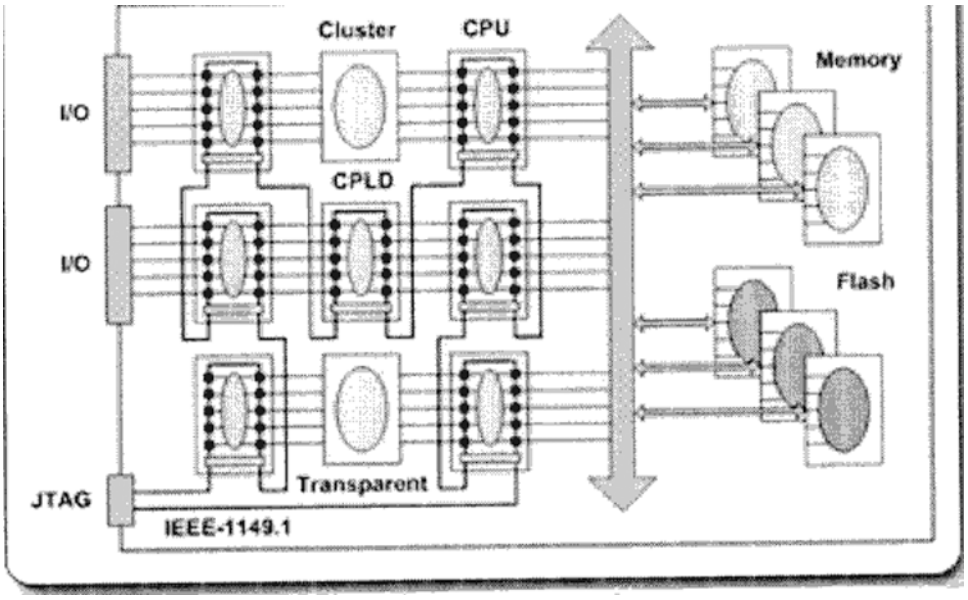


Figure D.3. Typical Board with Boundary-Scan Components

As shown in Figure D.3, a typical digital board with boundary-scan devices includes the following main components:

- Various boundary-scan components such as CPLDs, FPGAs, Processors, etc., interfaced via the boundary-scan path.
- Non-boundary-scan components (clusters).

- Various types of memory devices.
- Flash Memory components.
- Transparent components such as series resistors or buffers.

A typical boundary-scan test system is comprised of two basic elements: *Test Program Generation* (*TPG*) and *Test Execution* (*TE*). Generally, TPG requires the list of the *Unit Under Test* (UUT) and other files of the boundary-scan components. The TPG automatically generates test patterns that allow fault detection and isolation for all boundary-scan testable nets of the PCB. The TPG also creates test vectors to detect faults on the pins of non-scannable components such as clusters and memories that are surrounded by scannable devices.

SomeTPGs, like the Corelis boundary scan system, provide the user with a test coverage report, that allows the user to focus on the non-testable nets and determine what additional means are needed to increase the test coverage. Test programs are generated in seconds. For example when Corelis ScanPlusExpressTPG was used, it took a 200 MHz Pentium PC eight (8) seconds to generate an interconnect test for a UUT with 4,090 circuits (with17,500 pins). This generation time includes lists and all other input files processing, as well as test pattern file generation.

The TE tool provides means for executing boundary-scan tests and perform in-circuit-programming in a pre-planned specific order called a test plan. Test vectors files, which have been generated using the TPG, are automatically applied to the UUT and the results are compared to the expected values. In case of a detected fault, the system diagnoses the fault and lists the failures as depicted in Figure 6. Different test plans may be constructed for different UUTs. Tests within a test plan may be re-ordered, enabled or disabled, and unlimited different tests can be combined into a test plan. Corelis TE tool known as ScanPlus Runner also includes a test executive that is used to develop a test sequence or test plan from various independent sub tests. These sub tests can then be executed sequentially as many times as specified or continuously if desired. A sub test can also program CPLDs and flash memories.

One of the most important benefits of boundary-scan is during the production test phase. Automatic test program generation and fault diagnostics using Boundary-Scan (JTAG) software products and the requirements for no expensive fixtures results in a very economical test process. For products that contain edge connectors and digital interfaces that are not visible from the boundary-scan chain, boundary-scan vendors offer a family of boundary-scan controllable I/Os that provide a low cost alternative to expensive digital pin electronics.

D.5 Field Service Boundary-Scan Applications

Boundary-scan does not terminate with the shipping of a product. Periodic software and hardware updates must be performed remotely. For example, Flash ROM updates and reprogramming of programmable logic becomes a necessity. Service centers that normally would not want to invest in special support equipment to support a product, now have an option of using a standard PC or lap-top for boundary-scan testing. A simple PC based boundary-scan controller can be used for all tasks and also be used as a supplementary fault diagnostic system, using the exact same test vec-

tors that were developed during the design and production phase. This concept can be taken one step further by allowing an embedded processor access to the boundary-scan chain. This allows diagnostics and fault isolation to be performed by the embedded processor. The same diagnostic routines can be run as part of a power-on self-test procedure.

References and Suggestions for Further Study

1. Xilinx-ABEL Design Software Reference Manual, Data I/O Corp.
2. The ISP Application Guide and CPLD Data Book, Application Note XAPP075, Xilinx Inc., San Jose, CA.
3. D. Van den Bout, "Xilinx FPGA Student Manual", Prentice Hall, Englewoods Cliff, NJ.
4. R. Katz, "Contemporary Logic Design," Benjamin/Cummings Publ. Comp., Redwood City, CA.
5. J. Wakerly, "Digital Design," Prentice Hall, Englewoods Cliff, NJ.
6. Xilinx Foundation Series, On-line documentation, Xilinx, San Jose, CA.
7. Cadence Design Systems, Inc., *Verilog-XL Reference Manual.*
8. Open Verilog International (OVI), *Verilog HDL Language Reference Manual (LRM)*, 15466 Los Gatos Boulevard, Suite 109-071, Los Gatos, CA 95032.
9. Sternheim, E. , R. Singh, Y. Trivedi, R. Madhaven and W. Stapleton, Digital Design and Synthesis with Verilog HDL, published by Automata Publishing Co., Cupertino, CA.
10. Thomas, Donald E., and Philip R. Moorby, The Verilog Hardware Description Language, Second edition, published by Kluwer Academic Publishers, Norwell MA, ISBN 0-7923-9523-9.
11. Bhasker, J., A Verilog HDL Primer, Star Galaxy Press, 1058 Treeline Drive, Allentown, PA 18103, ISBN 0-9656277-4-8.
12. Wellspring Solutions, Inc., *VeriWell User's Guide 1.2*.
13. The IEEE Std 1149.1-1990 - Test Access Port and Boundary-Scan Architecture, and the Std 1149.1-1994b - Supplement to IEEE Std 1149.1-1990, are available from IEEE Inc., 345 East 47th Street, New York, NY 10017, USA, www.ieee.org

Index

Numerics

@alternate Directive in ABEL A-6
@radix Directive in ABEL A-7
@Radix in ABEL A-5
@standard Directive in ABEL A-7
2*421 code 4-3
4256V CPLD 11-14
4-to-16 decoder 7-29

A

ABEL A-1
abstract data types in Verilog C-11
accessing a set in ABEL A-8
accumulator register 10-4
ACK 4-6
Active-Low Declarations in ABEL A-23
adaptive logic module 11-41
adaptive look-up tables 11-41
addition of binary numbers 2-1
addition of hexadecimal numbers 2-5
addition of octal numbers 2-2
Advanced Boolean Equation Language A-1
ALM 11-41
alphanumerics 4-1
ALUT 11-41
American National Standards Institute 4-5
American Standard Code for Information Interchange (ASCII) 4-13
analog computer 10-1
analog-to-digital conversion 10-2
AND operation 5-1
ANSI 4-5
Antifuse-based FPGA 11-37
architectural declaration in VHDL B-29
architecture bodies in VHDL B-29
architecture declaration in VHDL B-2
argument in Verilog C-10
Arithmetic / Logic Unit 10-3
arithmetic operators in ABEL A-12
array aggregate in VHDL B-10
array in VHDL B-9
ascending ranges in VHDL B-6
assertion statement in VHDL B-19
Assignment operators in ABEL A-13
assignment statement in VHDL B-15
associative laws 5-2
asynchronous binary ripple counter 8-19
asynchronous flip-flop 8-2
asynchronous transfer 8-27
AT40KAL 11-40
attributes in VHDL B-13
Axcelerator 11-43
axioms 5-1

B

base 10 number 1-1, 1-3, 1-9
base 16 number 1-2, 1-9
base 2 number 1-1, 1-9
base 8 number 1-2, 1-9
base of a number 1-1
base2dec(s,b) 2-4
basic input/output system 10-3
basic input/output system 9-8
BCD 4-1
behavioral constructs in Verilog C-2
behavioral specification in Verilog C-7
behavioral description in VHDL B-1
BEL 4-6
Binary 1
binary addition 2-1
binary arithmetic operators in Verilog C-11
Binary Coded Decimal 4-1
binary number 1-1
binary-to-decimal number conversion 1-3, 1-11
binary-to-hexadecimal number conversion 1-8
binary-to-octal decoder 7-29
binary-to-octal number conversion 1-7, 1-11
BIOS 9-8, 10-3
bistable multivibrators 8-1
bit strings in VHDL B-5
block in VHDL B-30
blocks in VHDL B-1
Boolean algebra 5-1
Bose-Chauduri codes 4-10
boundary-scan board D-4
Boundary-scan D-1
Boundary-Scan Description Language D-1
box in VHDL B-10
BS 4-6
BSDL D-1
buried macrocell 11-13

C

cache memory 9-9
CAN 4-7
canonical form 6-5
carry flip flop 10-6
CarryConnect 11-48
Case statement in ABEL A-20
cell sensor 9-8
central processing unit 10-3
central switch matrix 11-13
CGROM 9-6
character generator ROM 9-6
characteristic of a number 3-2
characteristic table 8-2
checksum 4-10
chip 10-4
CLB 11-37
clocked circuit 8-14
cluster 11-45
coefficients of a number 1-1
combinational logic circuit 7-1
combinational logic circuit 8-1
combinational operator in ABEL A-13
comments in ABEL A-3
comments in VHDL B-4
commutative laws 5-2
complementation 5-1
complements of numbers 2-6
Complex Programmable Logic Devices 11-6
computer hardware 10-1
computer interfacing 10-1
computer software 10-2
Concat block 11-56
concurrent signal assignment in VHDL B-38
conditional signal assignment in VHDL B-38
configurable logic block 11-37
conjunction 5-1
constant in VHDL B-12
continuous assignment statements in Verilog C-17
control codes 4-5
control gate 9-7
control statements in Verilog C-15
control unit 10-3
Convert block 11-56
CPLD 11-6
CPU 10-3
CR 4-7
CRC code 4-10
Cyclic codes 4-9
cyclic redundancy check code 4-10

D

data flip-flop 8-4
data types in Verilog C-9
DC1 4-7
DC2 4-7
DC3 4-7
DC4 4-7
dec2hex(d) MATLAB function 3-1, 3-2
decimal number 1-1, 2-1
decimal-to-BCD encoder 7-26
decimal-to-binary number conversion 1-4, 1-11
decimal-to-hexadecimal number conversion 1-6, 1-9

decimal-to-octal number conversion 1-5, 1-9
declarations in ABEL A-3
decoder circuit 7-28
deferred constants in VHDL B-12
DEL 4-8
delimiters in Verilog C-6
DeMorgan's theorems 5-3
demultiplexer 7-36
descending ranges in VHDL B-6
design entity in VHDL B-1
design file in VHDL B-42
design in VHDL B-1
design library in VHDL B-42
design under test in VHDL B-28
device in ABEL A-3
digital computer 10-1
digital-to-analog conversion 10-3
DirectConnect 11-48
Directives in ABEL A-6
disjunction 5-1
distributive laws 5-2
DLE 4-7
don't care condition 7-20
don't care in ABEL A-15
dot extensions in ABEL A-21
double precision 3-7
DPA 11-41
DRAM 9-3
driver for the signal in VHDL B-37
D-type flip-flop 8-4
DUT in VHDL B-28
dynamic phase alignment 11-41
dynamic RAM 9-3

E

EBCDIC 4-8
edge triggering 8-8
electrically-erasable programmable read-only memory 9-8
elements in VHDL B-4
EM 4-7
encoder circuit 7-23
encoding 4-1
end around carry 2-11, 2-15
end carry 2-10, 2-14
end in ABEL A-3
ENQ 4-6
EnReg 11-47
entity in VHDL B-1
enumerations in VHDL B-6
EOT 4-6
equality comparator 7-32
equationsin ABEL A- 3, A-14
error correcting codes 4-9
error detecting codes 4-9
error polynomial 4-10
ESC 4-7
ETB 4-7
ETX 4-6
event in Verilog in Verilog C-22
excess-3 code 4-2
expanders 11-8
exponent 3-2
expression in VHDL B-14
Extended ASCII Character Set 4-8
Extended Binary Coded Decimal Interchange Code 4-8

F

FastConnect 11-48
FF 4-7
field programmable device 11-1
field programmable gate array 11-36
field programmable system chips 11-14
files in VHDL B-6
firmware 9-6
flag 11-54
flash memory 9-8
Flash370 11-20
flip-flops 8-1
floating gate 9-7
Floating point arithmetic 3-2
fork control in Verilog C-23
FPD 11-1
FPGA 11-36
FPSC 11-14
fraction of a number 3-2
FS 4-7
full adder 10-4
full adder 7-43
full subtractor 7-45
function subprograms in VHDL B-20
functional description in VHDL B-1
Functions in Verilog C-21

G

gateway in and out blocks 11-52
general purpose computer 10-1
generated carry 10-9
generator polynomial 4-10,
generic logic block 11-15
global routing pool 11-15
Gray code 4-4
GS 4-8

H

half adder 7-42
half subtractor 7-42
halt 10-3
hardware 10-1
Hardware Description Language A-1
HDL A-1
hexadecimal addition 2-5
hexadecimal number 1-2, 1-9
hexadecimal subtraction 2-5
hexadecimal-to-binary number conversion 1-8, 1-11
hexadecimal-to-decimal number conversion 1-3, 1-11

I

I/O control blocks 11-7
identifiers in Verilog C-7
identifiers in VHDL B-4
identity laws 5-2
IEEE Standard 754 3-3
IEEE Std 1532 11-7
if statement in VHDL B-16
If-Then-Else in ABEL A-19
in system programmability 11-7
indexing a set in ABEL A-8
indexing in ABEL A-8
inertial delay in VHDL B-35
input unit 10-2
InReg 11-47
integer in VHDL B-7
integer type in VHDL B-6
integrated circuit 10-4
interrupt 10-3
inversion 5-1
ispLSI 11-14
ispMACH 11-14
ispXPGA 11-42

J

JK-type flip-flop 8-5
Johnson counter 8-33
join control in Verilog C-23
Joint Test Action Group D-1
JTAG D-1

K

Karnaugh maps 7-12
K-maps 7-12

L

L1 cache 9-9
L2 cache 9-9
L3 cache 9-9
LAB 11-7
Lattice 3000, 4000, 5000 11-14
Lattice ispMACH 11-14
Lattice pLSI, ispLSI 11-14
Level 1 cache 9-9
Level 2 cache 9-9
Level 3 cache 9-9
lexical conventions in Verilog C-6
LF 4-7
library unit in VHDL B-42
literal characters in VHDL B-5
literal numbers in VHDL B-4
literal strings in VHDL B-5
load command 8-28
logic array blocks 11-7

logic block 11-16
logic Description in ABEL A-14
logic element 11-37
logic operations 5-1
logical name in VHDL B-42
logical operators in ABEL A-11
logical operators in Verilog C-12
logical product 5-1
logical sum 5-1
look up table 9-5
look-ahead carry 10-9
look-up table 11-37
loop statements in VHDL B-17
LUT 11-37

M

MAC 11-59
Mach 1 11-12
Mach 2 11-13
Mach 3 11-13
Mach 5 11-12
mantissa 3-2
mask-programmable 11-2
master/slave flip-flop 8-8
maxterms defined 6-2
memory stick 9-8
memory stick duo media 9-8
memory unit 10-3
message polynomial 4-10
microcomputer 10-3
microprocessor 10-3
minterms defined 6-1
module in ABEL A-3
module in Verilog C-2
module in VHDL B-1
modulo-2 addition 4-11
modulo-2 subtraction 4-11
Muller-C element in VHDL B-38
multiplexer 7-36
multiply-accumulate 11-59

N

NAK 4-7
natural subtype in VHDL B-10
negation 5-1
negation laws 5-2
negative pulse 8-7
negedge in Verilog C-22
nines-complement of a number 2-7, 2-14
node in ABEL A-5
non-blocking assignment in Verilog C-18
non-standard form 6-7
non-volatile memory 9-9
normalized 3-2, 3-9
NOT operation 5-1
NUL 4-5
null statement in VHDL B-19
number conversions 1-3
numbers in ABEL A-5
numbers in Verilog C-6

O

object in VHDL B-12
octal addition 2-3
octal number 1-2, 1-9
octal numbers 2-2, 2-14
octal subtraction 2-3
octal-to-binary encoder 7-24
octal-to-binary number
conversion 1-7, 1-11
octal-to-decimal number
conversion 1-3, 1-11
ones-complement of a number 2-9, 2-14
operating system 10-2
operator priorities in ABEL A-13
operators in ABEL A-11
operators in Verilog C-11
optimized reconfigurable cell array 11-42
OR operation 5-1
ORCA 11-42
other declarations in ABEL A-4
output unit 10-3
OutReg 11-47
overflow 10-6
overflow 11-54
overloading in VHDL B-23

P

package in VHDL B-24
page file 9-10
PAL 11-5
parallel adder 10-4
parallel expanders 11-9
parallel transfer 8-28
parity bit 4-8
pass gate 11-40
pause 10-3
peripherals 10-1
PFU 11-42
physical data types in Verilog C-9
physical type in VHDL B-7
PIA 11-7, 11-10
PIM 11-16
pin declarations in ABEL A-4
PLA 11-1
Place and Route Report 11-60
PLD A-1
pLSI 11-14
pointers in VHDL B-6
ports in VHDL B-1
posedge in Verilog in Verilog C-22
positive pulse 8-7
positive subtype in VHDL B-10
Post-Place and Route
Timing Report 11-60
postulate 5-1
precedence 7-1
precedence of operators in Verilog C-14
primary library units in VHDL B-42
primitives in Verilog C-7
procedural assignments in Verilog C -8
procedure subprograms in VHDL B-20
process in VHDL B-35
product of maxterms 6-4
product term allocator 11-17
product term steering 11-17
product terms 6-7
product-of-sums 6-7
programmable array logic 11-5
programmable function unit 11-42
programmable interconnect array 11-7
programmable interconnect matrix 11-16
programmable logic array 11-1
Programmable Logic Devices A-1
programmable read-only memory 9-6
projected output waveform in VHDL B-34
PROM 9-6
proof by perfect induction 5-3
propagated carry 10-9
property statements in ABEL A-23

Q

quantization 11-52
quotient polynomial 4-10

R

radix 1-1, 1-9
RAM 9-1
random access memory 9-1
read command 8-28
read-only memory 9-3
real in VHDL B-8
redundancy laws 5-2
register 8-27
Register Transfer Level C-1
registered operator in ABEL A-13
Reinterpret block 11-57
relational operators in ABEL A-12
remainder polynomial 4-10
repetition statements in Verilog C-16
reset direct 8-4
reset direct in ABEL A-17
ring counter 8-32
ring oscillator 8-35
ripple counter 8-21
ROM 3, 11
round 11-52
RS 4-8
RTL C-1

S

SAM 9-1
saturate 11-54, 11-62
scalar types in VHDL B-6
scratch-pad memory 9-10
sea-of-modules architecture 11-43
secondary library units in VHDL B-42
select gate 9-7
selected signal assignment in VHDL B-40
selection statements in Verilog C-15
self-complementing code 4-2
sequential circuit 8-1

serial access memory 9-1
serial adder 10-5
set direct 8-4
set in ABEL A-7
Set Operations in ABEL A-9
Set-Reset (SR) flip-flop 8-1
seven segment decoder/driver 7-51
seven segment display 7-51
shareable expanders 11-9
shift register 8-29
SI 4-7
sign magnitude 3-1, 3-10
signal assignments in VHDL B-3
simple programmable logic devices 11-6
single precision 3-3
slice 11-37
Slice block 11-57
SO 4-7
software 10-2
SOH 4-5
SPLD 11-6
SR flip-flop 8-1
SRAM 9-3
SRAM-based FPGA 11-36
standard form 6-7
standard in VHDL in VHDL B-43
standard products 6-1
standard sums 6-2
state diagram in ABEL A-18
state table 8-14
state_reg in ABEL A-19
state_value in ABEL A-19
static RAM 9-3
std in VHDL in VHDL B-43
stored program 10-1
string in ABEL A-3
string in Verilog C-7
structural specification in Verilog C-7
STX 4-5
SUB 4-7
subprograms in VHDL B-20
subroutine 10-2
subtraction of hexadecimal numbers 2-5
subtraction of octal numbers 2-3
subtraction with nines-complement 2-11
subtraction with ones-complements 2-11
subtraction with tens-complement 2-10
subtraction with twos-complement 2-10
sum of minterms 6-4
sum terms 6-7
sum-of-products 6-7
supercluster 45
Symbols
SYN 4-7
synchronous circuit 8-14
synchronous flip-flop 8-3
synchronous transfer 8-27
system tasks in Verilog C-19

T

T flip-flop 8-6
TAB 4-7
TE in boundary scan D-5
tens-complement of a number 2-7, 2-14
test execution in boundary-scan D-5
test generator in VHDL B-28
Test vectors in ABEL A-22
thrashing 9-10
three-phase clock generator 8-44
time in VHDL B-8
timing control in Verilog C-22
timing diagrams 7-11
title in ABEL A-3
toggle flip-flop 8-6
tokens in Verilog C-6
TPG in boundary-scan D-5
transfer command 8-27
transition table 8-11
transport delay in VHDL B-34
truncate 11-52
truth table 5-3
truth tables in ABEL A-15
twos-complement of a number 2-8, 2-14

U

unit under test in boundary-scan D-5
unnormalized 3-3, 3-9
US 4-8

V

valid bit input 11-55
variable in VHDL B-12
Verilog A-1
VeriWell C-1
Vern diagram 6-1
VHDL A-1
VHSIC Hardware Description Language A-1
virtual memory 9-9
volatile memory 9-1
VT 4-7

W

wait control in Verilog C-22
weighted code 4-1
with statement in ABEL A-20
work in VHDL B-42
wrap 11-54